Sami Tiilikainen

Web-sovelluksen toimintalogiikka palvelimelta käyttäjän selaimeen

Sähkotekniikan korkeakoulu

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 17.12.2012.

Työn valvoja:

Prof. Jukka Manner

Työn ohjaaja:

DI Markus Ahonen

Aalto-yliopisto
Sähkötekniikan korkeakoulu

AALTO-YLIOPISTO
SÄHKOTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Sami Tiilikainen

Työn nimi: Web-sovelluksen toimintalogiikka palvelimelta käyttäjän selaimeen

Päivämäärä: 17.12.2012 Kieli: Suomi Sivumäärä:7+69

Tietoliikenne- ja tietoverkkotekniikan laitos

Professuuri: Tietoverkkotekniikka Koodi: S-38

Valvoja: Prof. Jukka Manner

Ohjaaja: DI Markus Ahonen

Perinteisen web-sovelluksen esityslogiikka sijaitsee palvelinsovelluksessa ja sovellusta käytetään sivukokonaisuuksia synkronisesti lataamalla. Rikkaissa internetsovelluksissa tiedonsiirto palvelimen ja käyttöliittymän välillä on asynkronista ja vasteet käyttäjälle perinteistä web-sovellusta nopeampia. Tässä työssä selvitetään miten web-sovelluksen toimintalogiikkaa voidaan hajauttaa palvelimelta käyttäjien selaimiin ja miten se vaikuttaa sovelluksen suorituskykyyn.
JavaScript-selainsovelluksesta saadaan laadukas ja ylläpidettävä hyvän arkkitehtuurin ja suunnittelumallien avulla. Arkkitehtuurin hyvänä perustana toimivat laadukkaat JavaScript-kirjastot. Selainsovelluksen suorituskykyyn vaikuttavat tietoliikenne, sivun muodostus selaimessa ja käyttöliittymäsovelluksen logiikan suorittaminen selaimessa. Suorituskykyä voidaan parantaa käyttämällä tehokkaaksi havaittuja optimointikeinoja ja välttämällä hitaita toimintoja. Kirjallisuuskatsauksessa selvitettyjä optimointikeinoja kokeiltiin kahdessa tapaustutkimuksessa ja ne osoittautuivat käytännössä toimiviksi.

Avainsanat: internetsovellus, web-sovellus, html, html5, css, dom, ajax, comet, javascript, dom, rest, arkkitehtuuri, suunnittelumalli, sovelluskehys, sovelluskirjasto, mvc, json, selain

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Sami Tiilikainen

Title: Web application logic from server to client browser

Date: 17.12.2012 Language: Finnish Number of pages:7+69

Department of Communications and Networking

Professorship: Networking Technology Code: S-38

Supervisor: Prof. Jukka Manner

Advisor: M.Sc. (Tech.) Markus Ahonen

In traditional web applications presentation logic is located at server side applications and pages are loaded asynchronously. In rich internet applications the communication between server and user interface is asynchronous, providing faster responses to the user. This thesis discusses on distributing the application logic from server to user browsers and how it affects the application performance.
High quality and maintainability in JavaScript browser applications can be achieved with good architecture and use of design patterns. High quality JavaScript libraries serve as a base for good architecture. Web application performance is impacted by data transfer, rendering in browser and executing user interface logic in browser. Performance can be improved using recommended guidelines. The optimizing techniques presented in literature review are proven in two case studies.

Keywords: ria, html, html5, css, dom, ajax, comet, javascript, dom, rest, architecture, pattern, framework, library, mvc, json, browser

Esipuhe

Tämän työn teki mahdolliseksi työnantajani HiQ Finland Oy sekä työn tilaaja MTV Oy. Haluan kiittää työn mahdollistamisesta HiQ:n Jukka-Petri Sahlbergia ja MTV:n Marianna Chekurovaa.

Kiitän professori Jukka Manneria työn valvonnasta ja Markus Ahosta työn ohjaamisesta. Suuri kiitos kaikesta avusta ja hyvistä ideoista jotka veivät työtä eteenpäin! Kiitos myös avopuolisolleni Johanna Saarelle kannustamisesta työn eri vaiheissa.

Otaniemi, 17.12.2012

Sami Tiilikainen

Sisällysluettelo

Lyhenteet

AD	Active Directory
AJAX	Asynchronous Javascript and XML
AMD	Asynchronous Module Definition
API	Application Programming Interface
CRUD	Create/Read/Update/Delete
CSS	Cascading Style Sheets
DNS	Domain Name System
DOM	Document Object Model
EIS	Enterprise Information System
EJB	Enterprise Java Bean
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IIS	Internet Information Services
IP	Internet Protocol
JavaEE	Java Platform Enterprise Edition
JIT	Just-In-Time
JITC	Just-In-Time Compiler
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JSP	Java Server Pages
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
REST	Representational State Transfer
RIA	Rich Internet Application, Rikas internetsovellus
SGML	Standard Generalized Markup Language
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Unified Resource name
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
WS	Web Service
WWW	World Wide Web
XHR	Xml Http Request
XML	Extensible Markup Language

Luku 1

Johdanto

Tässä työssä tutkitaan web-sovellustekniikoiden kehitystä ja selaimella käytettävän web-sovelluksen esitys- ja toimintalogiikan viemistä palvelimelta käyttäjän selaimeen. Perinteisen web-sovelluksen tapauksessa sovelluslogiikasta vastaa palvelinsovellus, mutta uusimpien tekniikoiden avulla monimutkaistakin sovelluslogiikkaa on mahdollista suorittaa palvelimen sijaan käyttäjän selaimessa. Ennen uusien tekniikoiden käyttöönottoa on kuitenkin tärkeää selvittää miten erot vaikuttavat sovelluskehitykseen ja sovelluksen toimintaympäristöön: käyttäjän laitteistoon, palvelinympäristöön ja verkkoyhteyksiin.

Perinteisen web-sovelluksen ongelmia ovat alhainen vuorovaikutuksen taso ja kaiken sovelluslogiikan suorittaminen keskitetysti palvelinympäristössä. Sovelluksen tarjoamiseksi tarvitaan riittävästi palvelinresursseja, jotta kaikille käyttäjille voidaan tarjota tarvittavat laskentaresurssit ja kohtuullisen nopeat vasteet. Vaikka palvelinresursseja voitaisiin lisätä rajattomasti, perinteisen web-sovelluksen käyttöliittymästä ei saataisi yhtä nopeasti reagoivaa kuin työpöytäsovelluksien käyttöliittymät. Kun sovelluslogiikkaa siirretään palvelimelta selaimeen, kohdataan uusina haasteina selainsovelluksen arkkitehtuuri ja suorituskyky. Sovelluksen laadun ja elinkaaren kannalta on oleellista selvittää, voidaanko JavaScript-kielellä toteuttaa laaja ja ammattimainen käyttöliittymäsovellus, joka on arkkitehtuurillisesti yhtä selkeä ja ylläpidettävä kuin aiemmat palvelinpuolen sovellukset. Lisäksi web-sovelluksen suorituskykyyn vaikuttavat tekijät muuttuvat, jolloin sovelluksen suorituskyvyn optimointiin tarvitaan uudenlaisia keinoja.

Rikkaan internetsovelluksen toteutustekniikoiden avulla voidaan tavoitella parempaa käyttäjän laitteiston resurssien hyödyntämistä ja parempaa vuorovaikutusta käyttäjän ja sovelluksen välillä. AJAX-tekniikoita hyödyntäen käyttöliittymän ja palvelinsovelluksen välinen tietoliikenne voi olla asynkronista: tietoa voidaan siirtää sivukokonaisuuksien sijaan pienissä osissa käyttäjän huomaamatta. JavaScript-ohjelmointikieli, yhdessä nykyaikaisten selainrajapintojen kanssa, mahdollistaa käyttäjän laitteistoresursseja hyödyntävän sovelluslogiikan suorittamisen käyttäjän selaimessa. Lisäksi uusimmat HTML5 ja CSS -tekniikat mahdollistavat graafisesti rikkaat käyttöliittymät.

Sovelluksen rakenteen suunnittelussa auttavat koetellut sovelluskehityksen suunnittelumallit sekä laadukkaat kirjastot ja sovelluskehykset. Suorituskykyä voidaan

parantaa tehostamalla verkkoliikennettä, sivun muodostamista selaimessa ja JavaScript-sovelluskoodia. Tehostaminen saavutetaan noudattamalla tehokkaaksi havaittuja käytäntöjä ja välttämällä suorituskyvyltään hitaiksi todettuja toimintoja.

Tämän tutkimuksen tavoitteena on toimia kattavana ohjeistona rikasta internetsovellusta suunnitteleville ja auttaa saavuttamaan sekä arkkitehtuurin että suorituskyvyn osalta korkealaatuisen lopputuloksen. Työn alkuosa tutustuttaa lukijan uusimpiin tekniikoihin, ja loppuosa auttaa kehittäjää välttämään useita suorituskyky- ja arkkitehtuuriongelmia.

Työn alussa tutkitaan web-sovellusten kehitystä yksinkertaisista HTML-dokumenteista ja palvelinpuolen sovelluksista nykyaikaisiksi rikkaan vuorovaikutuksen selainsovelluksiksi. Luvussa 2 esitellään web-sovellusten perustekniikat ja kerrotaan mitä tarkoittavat esimerkiksi HTTP, URL, HTML ja CSS. Luvussa 3 esitellään rikkaan internetsovelluksen toteuttamisen liittyvät tekniikat ja avataan termit kuten AJAX, JavaScript, JSON, DOM, REST, HTML5 ja PUSH.

Rikkaan internetsovelluksen perusteiden jälkeen luvussa 4 syvennytään JavaScript-sovellusten arkkitehtuuriin ja suunnittelumalleihin. JavaScript-sovellukset ovat kypsyneet täysipainoisiksi sovelluksiksi siinä missä muillakin ohjelmointikielillä toteutetut sovellukset, ja siten niihin pätee sama arkkitehtuurin ja huolellisen suunnittelun tarve. Luokkamäärittelyihin perustuviin, käännettäviin ja palvelinympäristössä ajettaviin kieliin verrattaessa JavaScript tuo joitakin muutoksia web-sovelluskehitykseen. Tarjolla on hyödyllisiä suunnittelumalleja, kirjastoja ja sovelluskehyksiä, jotka auttavat keskittymään sovelluskehityksessä oleelliseen.

Web-sovellusten suorituskykyä käsitellään luvussa 5. Käyttäjän kokema sovelluksen suorituskyky muodostuu verkkoliikenteen, palvelinprosessoinnin, sivun muodostamisen ja selainsovelluksen sovelluslogiikan suorituskyvystä. Kaikkien osa-alueiden suorituskykyä voidaan parantaa noudattamalla työssä esitettäviä hyviä käytäntöjä. Luvussa 6 suoritetaan mittauksia, joilla voidaan vahvistaa optimointitekniikoiden ja asynkronisen toimintatavan toimivuus käytännössä.

Luku 2

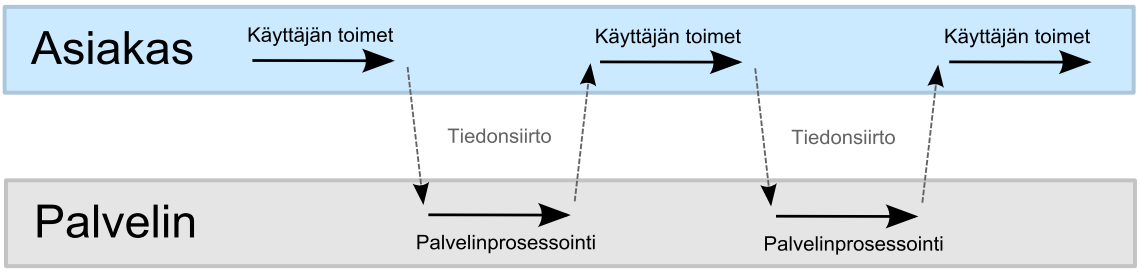
Perinteinen web-sovellus

Perinteisen web-sovelluksen perustekniikat tarjoavat perustan rikkaille internetsovelluksille. Tässä luvussa esitellään perinteiseen web-sovellukseen liittyviä protokollia ja merkintäkieliä, jotka ovat edelleen käytössä myös rikkaissa web-sovelluksissa.

2.1 Perinteisen web-sovelluksen ominaispiirteitä

Perinteinen web-sovellus rakentuu sivunvaihdon käsitteen ympärille. Käyttäjän ja palvelimen välinen synkroninen malli esitetään kuvassa 2.1. Kun käyttäjä haluaa lähettää palvelimelle tietoa tai ladata uutta sisältöä, lähetetään uusi sivupyyntö ja sen mukana mahdollisesti käyttäjän syötteitä. Käyttäjän kerralla käsiteltävissä oleva sisältö perustuu sivupyynnön vastauksena sivulle muodostettuihin lomakkeisiin. Kun palvelin prosessoi käyttäjän syötteitä ja muodostaa sivupyyntöön vastausta, ei sovelluksen käyttöliittymä ole käytettävissä. Käyttäjä joutuu siten odottamaan palvelinta ja sovelluksen käyttökokemus on katkonainen.

Perinteisen web-sovelluksen verkkoliikenne muodostuu sivun ja siihen liittyvien tiedostojen latauksesta purskeena. Verkkoliikenteen näkökulmasta sovelluksen käytössä on useiden sekuntien tai minuuttien katkoksia, kun käyttäjä lukee sivua ja täyttää lomaketietoja. Vaikka vain osaa sivun lomaketiedosta olisi käsitelty, tyypillisesti kaikki lomaketiedot lähetetään palvelimelle. Muokattuja tietoja tallenet-



Kuva 2.1: Perinteisen web-sovelluksen käyttäjän ja palvelimen välinen synkroninen kommunikointi. [20]

taessa joudutaan tyypillisesti tarkistamaan kaikki lomaketiedot ja vertaamaan niitä olemassa oleviin, jotta voidaan päätellä mikä tiedoista on uutta tai muuttunutta. Sivupyynnön raskauteen voidaan vaikuttaa lisäämällä tai vähentämällä yhden sivun sisältämän tiedon määrää ja siten pilkkomalla käyttäjän kerralla tekemää työtä pienempiin osiin. [9]

Perinteisen web-sovelluksen käyttöliittymä on visuaalisesti hyvin rajoittunut. Käyttöliittymä koostuu staattisesta ulkoasusta ja yksinkertaisesta hiiren liikkeisiin reagoimisesta. Vapaata piirtoaluetta, animointeja tai muuta monipuolista vuorovaikutteista reagointia ei ole.

Kaikki liikenne on pull-tyyppistä, eli kaikki toiminnot ovat asiakaslähtöisiä. Asiakas pyytää jotakin resurssia ja palvelin vastaa pyyntöön, mutta palvelin ei koskaan aloita vuoropuhelua. Jos tieto muuttuu taustalla sivun avaamisen ja sitä seuraavan tallennustoimenpiteen välissä, käyttäjä saa tästä tiedon vasta tallennuksen suorittavan sivupyynnön jälkeen, ja voi mahdollisesti menettää työnsä päällekkäisten muutosten kohdalla.

2.2 Protokollat ja tekniikat World Wide Webin takana

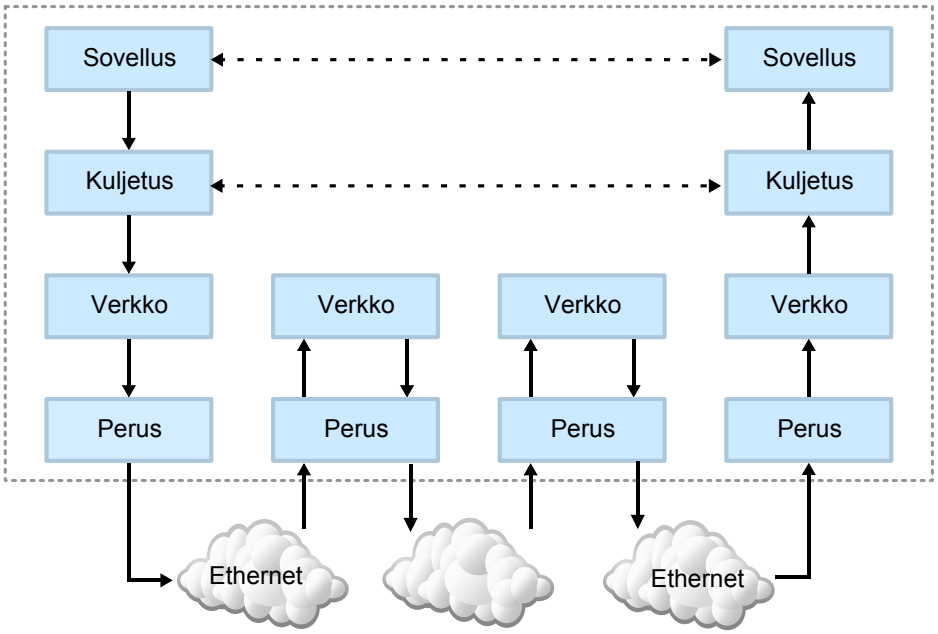
Web-sovelluksen käyttäjälle näkyvin osa on web-selain, sen näyttämät Hypertext Markup Language (HTML) -sivut ja niiden Uniform Resource Locator (URL) -osoitteet. Web-sivu koostuu yksinkertaisimmillaan HTML-sivun lisäksi siihen liittyvistä kuvista, Cascading Style Sheets (CSS) -tyyliohjeista ja selaimessa suoritettavasta JavaScript -sovelluskoodista. Selain käsittelee HTML-dokumenttia puumaisena tietorakenteena Document Object Model (DOM) -mallin mukaisesti. JavaScript mahdollistaa web-sovelluksen vuorovaikutuksen sivun DOM-rakenteisiin.

Protokollat

TCP/IP-viitemallin eri kerrokset esitetään kuvassa 2.2. Sivupyynnöt ja niiden vastauksena HTML-sivut liikkuvat käyttäjän (eng. user agent) ja palvelimen (eng. origin server) välillä Hypertext Transfer Protocol (HTTP) -protokollaa käyttäen. HTTP-protokolla sijaitsee TCP/IP-viitemallin korkeimmalla sovelluskerroksella. HTTP-viestit kuljettaa kuljetuskerroksen Transmission Control Protocol (TCP) -protokolla, joka vastaavasti toimii verkkokerroksen Internet Protocol (IP) -protokollan tarjoaman osoitteiston päällä. Kerrosmallin pohjalla peruskerros huolehtii tiedon fyysisestä välittämisestä laitteesta toiseen.

HTTP-liikenne perustuu asiakas-palvelin malliin. Asiakas, eli sovelluksen käyttäjä, aloittaa kommunikoinnin ja palvelin vain vastaa sille lähetettyihin pyyntöihin. HTTP-protokollan kaksi perustoimintoa ovat GET ja POST, joita käyttäen selain voi pyytää palvelimelta sivuja ja lähettää palvelimelle lomaketietoja. [16]

HTTP on tilaton protokolla, eli protokolla itse ei ylläpidä sovelluksen tilaa. HTTP-protokollan evästeet (eng cookie) mahdollistavat tilatiedon siirtämisen sivupyyntöjen ja vastausten yhteydessä, mutta tilatiedon ylläpito on täysin sovelluksen



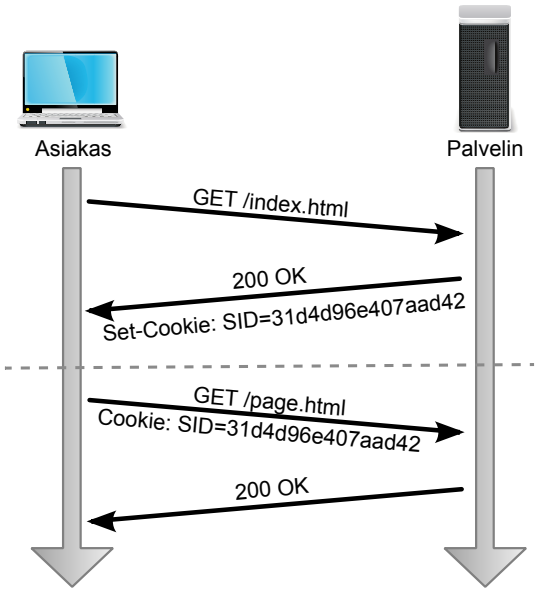
Kuva 2.2: TCP/IP-viitemalli.

vastuulla. Eväste on selainkohtaisesti käyttäjän kiintolevylle tallennettava avain-arvo pari, jossa arvon koko on korkeintaan 255 merkkiä ja neljä kilotavua. Palvelin luo evästeet ja lähettää ne HTTP-vastauksessa käyttäjän selaimelle. Selain tallentaa evästeen ja palauttaa sen jatkossa jokaisen sivupyynnön yhteydessä palvelimelle. Eväste on sivustokohtainen, ja aina kun käyttäjän selain lähettää pyyntöjä evästeelle määritellyn domain-nimen tai IP-osoitteen alueella oleviin URL-osoitteisiin, eväste lähetetään sivupyynnön mukana. Evästeiden avulla tallennettu tilatieto on saatavissa vain tietyn asiakaskoneen tietyllä selaimella, kutsuttaessa tietyn palvelimen sivuja. Evästeet mahdollistavat selaimen tunnistamisen (evästeen voimassaolon ajan), mutta eivät mahdollista tietyn käyttäjän luotettavaa tunnistamista. [39] [3]

Tyypillinen evästeiden käyttötapaus on tallentaa käyttäjän istuntoa kuvaava yksilöllinen tunniste selaimeen. Tunnisteen avulla voidaan muistaa sisäänkirjautunut käyttäjä useiden eri sivupyyntöjen ajan, ilman että käyttäjän tarvitsee lähettää käyttäjätunnusta ja salasanaa uudelleen saman istunnon aikana. Evästeelle annettavia lisätietoja ovat vanhenemisajankohta tiettynä päivämääränä (`Expires`), maksimiikä sekunteina (`Max-Age`), tiettyyn domain-osoitteeseen rajoittaminen (kuten alidomain `esimerkki.domain.com`), tiettyyn polkuun rajoittaminen, salattuun sisältöön rajoittaminen ja vain http-protokollaan rajoittaminen. [3]

Kuvassa 2.3 esitetään istuntotunnisteen asettaminen SID-nimiseen evästeeseen ja sen palauttaminen palvelimelle. Eväste voidaan asettaa palvelimelta asiakkaalle HTTP-protokollan `Set-Cookie` -otsikkotiedolla. Tämän jälkeen asiakkaan selain lähettää kaikissa sivupyynnöissä palvelimelle HTTP-otsikon `Cookie`.

Evästeet olivat pitkään ainoa tapa tallentaa web-sovellukseen liittyvää tietoa asiakastietokoneelle. HTML5-tekniikat ovat tuoneet kehittyneempiä vaihtoehtoja



Kuva 2.3: Evästeen asettaminen käyttäjän selaimeen ja sen palauttaminen palvelimelle.

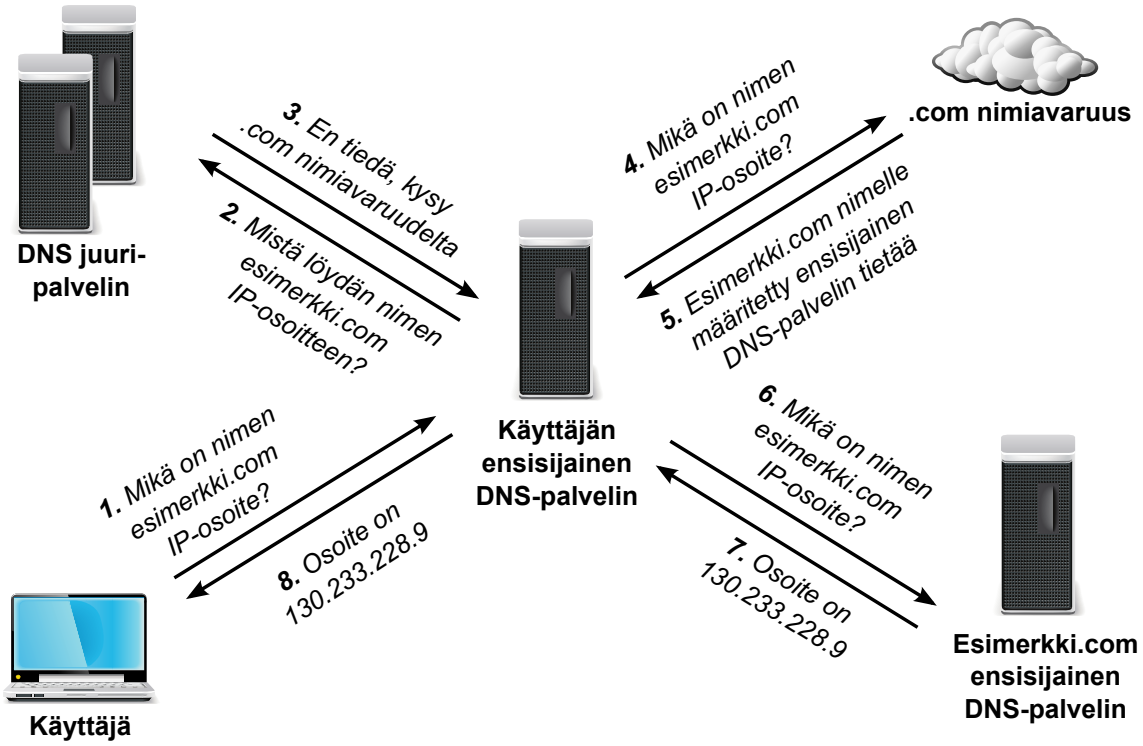
tiedon tallentamiseen käyttäjän selaimeen. Niitä käsitellään tarkemmin luvussa 3.3.

Osoitteet

Internetiin kytketyt laitteet käyttävät tiedonsiirrossa IP-protokollan osoitteita. Osoitteet ovat protokollan versiosta riippuen joko 32-bittisiä (IPv4) tai 128-bittisiä (IPv6) numerosarjoja, jotka esitetään yleensä ihmiselle luettavaan muotoon ryhmiteltynä. IP-osoitteet eivät tyypillisesti näy web-sivuston käyttäjälle asti. Sen sijaan loppukäyttäjälle näkyvät domain-nimet, jotka muunnetaan sivupyyntöjä tehtäessä IP-osoitteiksi nimipalvelun (Domain Name System, DNS) avulla. Esimerkiksi `google.fi` -domain-nimen takaa löytyvä palvelin sijaitsee IPv4-osoitteessa `74.125.227.127` ja IPv6-osoitteessa `2607:f8b0:4000:800::1017`. [32]

Kuva 2.4 esittää DNS-mallin osapuolet ja tiettyä domain-nimeä vastaavan IP-osoitteen selvittämiseksi tarvittavat kyselyt. Käyttäjän tietokone kysyy DNS-tietoja käyttäjän ensisijaiselta DNS-palvelimelta. Sen jälkeen käyttäjän ensisijainen DNS-palvelin aloittaa nimiselvitykset, mikäli nimi ei ole entuudestaan tuttu. Se kysyy tarkempaa tietoa DNS juuripalvelimelta ja saa vastauksena tiedon mistä nimiavaruudesta löytyy lisää tietoa osoitteesta. Esimerkki.com nimen tapauksessa osoitetietoa kysytään .com -nimiavaruudesta, mistä saadaan vastauksena kyseisen domain-nimen ensisijainen DNS-palvelin. Domain-nimen ensisijainen DNS-palvelin kertoo lopulta osoitetta vastaavan IP-osoitteen. Mikäli ensisijainen DNS-palvelin ei olisi tavoitettavissa, voitaisi tietoa pyytää saman tiedon omaavalta toissijaiselta palvelimelta. [32]

Uniform Resource Identifier (URI) on resurssit yksilöivä tunniste, jolla esitetään resurssin nimi tai sijainti. Uniform Resource Name (URN) on URI joka nimeää re-



Kuva 2.4: Domain-nimeä vastaavan IP-osoitteen selvittäminen DNS-kyselyillä. [32]

surssin yksikäsitteisesti, mutta jonka avulla ei ole tarkoitus löytää resurssia. Verkkosivujen avaamiseen käytettävä Uniform Resource Locator (URL) -osoite on URI joka paitsi yksilöi resurssin, myös kertoo resurssin sijainnin. Sivupyynnössä URL kertoo palvelimelle mitä resursseja käyttäjä haluaa hakea. URL-osoite yhdistää domain-nimen tai IP-osoitteen, tiedostopolun ja sovellukselle välitettävät parametrit yhdeksi osoitteeksi. Osoitteen syntaksi on muotoa:

```
skeema://domain:portti/polku?parametrit#kohdan_tunniste
```

Parametrit ovat avain-arvo pareja muodossa `avain=arvo`, erotettuna &-merkillä. Web-käytössä kuljetuskerroksen porttia ei yleensä tarvitse kirjoittaa osoitteeseen, vaan käytetään oletuksena HTTP-protokollalle varattua TCP-porttia 80. [5] [16]

Hypertext Markup Language -kuvauskieli

HyperText Markup Language (HTML) on standardoitu kuvauskieli, jonka avulla kuvataan web-sivun rakenne. HTML perustuu Standard Generalized Markup Language (SGML) -syntaksiin ja sille on Extensible Markup Language (XML) -syntaksia noudattava XHTML-vastine. [28]

HTML-määrittelyn on alun perin kehittänyt Tim Berners-Lee. HTML-määrittelyn versioita 2.0 ja 3.0 kehitti Internet Engineering Task Force (IETF) vuonna 1995, mutta vuonna 1997 esiteltyjen versioiden 3.2 ja 4.0 standardoinnista on vastannut World Wide Web Consortium (W3C). HTML-version 4.01 jälkeen W3C lo-

Listaus 2.1: HTML-elementti

```
<p id="attribuutin_arvo" class="attribuutin_arvo">
  Elementin tekstisisältö tulee aloitus- ja lopetustagien
      väliin. <br />
</p>
```

Listaus 2.2: Esimerkki HTML-sivun syntaksista [28]

```
<!DOCTYPE html>
<html>
  <head>
    <title>Esimerkkisivu</title>
  </head>
  <body>
    <h1>Korkeimman tason otsikko</h1>
    <p>Tekstisisältöä ja
      <a href="demo.html">linkki</a>.</p>
    <!-- Kommentti ei näy selaimen esittämässä
       dokumentissa -->
  </body>
</html>
```

petti sen kehittämisen ja siirtyi kehittämään uutta XML-pohjaista XHTML 1.0 -määrittelyä, joka ei ollut yhteensopiva aiemman HTML-määrittelyn kanssa. Selainvalmistajien yhteenliittymä Web Hypertext Application Technology Working Group (WHATWG) jatkoi HTML-kielen kehittämistä. Lopulta vuonna 2007 W3C keskeytti XHTML 1.0 -määrittelyn kehittämisen ja ryhtyi yhdessä WHATWG-ryhmän kanssa kehittämään yhteistä HTML5-määrittelyä. [42] [28]

HTML-dokumentti rakentuu listauksen 2.1 mukaisista elementeistä, jotka puolestaan muodostuvat aloitus- ja lopetustagista, niiden välisestä sisällöstä sekä aloitustagin attribuuteista. Elementillä ei tarvitse olla sisältöä, jolloin elementti voidaan lopettaa aloitustagin lopussa /-merkillä. Esimerkkinä listauksen 2.1 `br`-elementti. [42]

HTML-dokumentti on jaettu otsikkotietoihin (`head`-elementti) ja näkyvään sisältöosaan (`body`-elementti). Listauksessa 2.2 esitetään yksinkertaisen HTML-dokumentin rakenne.

Cascading Style Sheets -tyyliohjeet

HTML-dokumentin sisältö muotoillaan selaimessa sivulle määriteltyjen Cascading Style Sheets (CSS) -tyyliohjeiden mukaisesti. CSS mahdollistaa muotoilun irrotta-

misen HTML-dokumentista erillisiin tyylitiedostoihin. Tyylitiedosto koostuu useista tyylisäännöistä, jotka sisältävät valitsimen ja määrittelyn. Valitsimella valitaan mihin dokumentin sisältöön sääntö vaikuttaa ja määrittely on lista ominaisuuksista ja niiden arvoista. Tyyliohjeilla voidaan määrittää HTML-sivun elementille esimerkiksi sijainti, värit ja fontin muotoilu. Ennen CSS-tyyliohjeiden käyttöä kaikki HTML-sivun muotoilu oli määritelty suoraan HTML-dokumenttiin elementein ja attribuutein. [6]

Ensimmäisen tason CSS 1 -määrittely julkaistiin vuonna 1996 ja se sisälsi tekstin ja fontin muotoiluun ja värittämiseen liittyvät perusominaisuudet, sekä elementtien marginaalit, kehykset ja sijoittelun. Toisen tason CSS 2 -määrittely laajensi ensimmäistä tasoa uusilla sijoitteluun liittyvillä ominaisuuksilla vuonna 1998, ja sitä korjattiin myöhemmin CSS 2.1 -määrittelyllä. Toista tasoa on laajennettu moduleittain CSS 3 -määrittelyillä, joista ensimmäiset on julkaistu suosituksina vuonna 2011. [6] [8]

Pääsääntöisesti uudet CSS-määrittelyt laajentavat aiempaa määrittelyä, ja ovat siten yhteensopivia vanhojen määrittelyiden kanssa. Eri selaimissa on hieman erilaisia tulkintoja CSS-määrittelyistä ja sen vuoksi kehittäjät ovat kohdanneet yhteensopivuusongelmia. [6]

Listaus 2.3: Esimerkki CSS-tyylimäärittelyn syntaksista

```
valitsin [, valitsin2, ... ] {
  ominaisuus: arvo;
  [ominaisuus2: arvo2; ...]
}
```

Listaus 2.4: Esimerkki CSS-tyylimäärittelystä HTML-dokumentille

```
body {
  font-family: Arial, Verdana, Helvetica;
  background: white;
}
h1#uniikki-tunniste {
  font-size: 2em;
  color: green;
}
.perusteksti {
  margin: 1em;
  padding: 1em;
  color: black;
}
.perusteksti a,
.perusteksti a:link {
  color: blue;
}
```

2.3 Palvelimet ja taustajärjestelmät

Perinteisessä web-sovelluksessa palvelimet ja taustajärjestelmät vastaavat kaikesta toimintalogiikasta ja myös käyttöliittymän HTML-rakenteen muodostamisesta. Taustalla voi olla useita järjestelmiä, jotka kommunikoivat keskenään sivupyynnön vastauksen muodostamiseksi.

Palvelinsovellus vastaa toimintalogiikasta

Perinteisen web-sovelluksen toimintalogiikka painottuu palvelinpuolelle ja käyttäjän selain toimii yksinkertaisena valmiin HTML-dokumentin esittäjänä. Sovelluskoodia suoritetaan palveluntarjoajan sovelluspalvelimella ja sovellus palauttaa käyttäjälle HTTP-pyyntöjen ja URL-osoitteiden perusteella valmiita HTML-dokumentteja. Käyttäjä antaa syötteensä sovellukseen dokumenttien sisältämien HTML-lomakkeiden välityksellä, jonka jälkeen palvelin käsittelee syötteen ja palauttaa niiden perusteella jälleen valmiin HTML-sivun.

Käyttäjän selain esittää HTML-sivut annettujen tyylisääntöjen mukaisesti, mutta ei vaikuta sovelluslogiikkaan. Sivun muodostus koostuu tyypillisesti useista palvelinpuolen sovelluksen toiminnoista, ja tyypillisesti käyttäjä joutuu hetken odottamaan sivun muodostamista palvelimella.

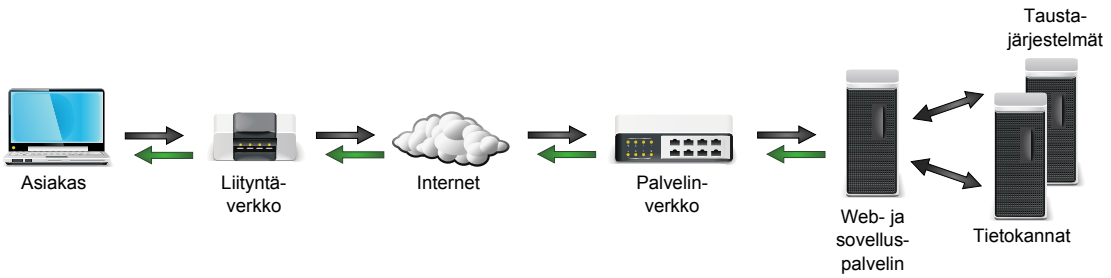
Yritysten www-palvelinsovellukset on usein kirjoitettu Java-kielellä hyödyntäen Java EE (Java Platform, Enterprise Edition) -ohjelmistokehitysalustaa. Java EE on määrittely, jossa sovelluslogiikka on jaettu toiminnallisuuden perusteella komponentteihin. Komponentit muodostavat sovellukselle kerroksittaisen rakenteen ja ne voidaan hajauttaa useille laitteille.

- Asiakaskerroksessa on käyttäjän web-selain tai Java-asiakasohjelma.
- Web-kerroksessa asiakkaaseen on yhteydessä JavaServer Pages (JSP) -sivut tai Java servlet.
- Bisneskerroksessa Enterprise Java Beans (EJB) -pavut vastaavat bisneslogiikasta.
- Kerrosrakenteen pohjalla Enterprise Information System (EIS) -kerroksessa sijaitsee tietokannat ja muut järjestelmät joihin sovellus on yhteydessä.

Vastaavia palvelinpuolen tekniikoita ovat esimerkiksi Microsoftin ASP.NET sekä harrasteprojekteista suureen suosioon kasvaneet PHP ja Ruby on Rails.

Sovelluksen toimintaan liittyvät palvelimet

Kuva 2.5 esittää asiakkaan sivupyyntöön liittyvät järjestelmät. Asiakkaan sivupyyntö lähtee asiakkaan päätelaitteelta liityntäverkon kautta internetiin reititettäväksi, ja päätyy lopulta palveluntarjoajan palvelinverkkoon ja web-palvelimelle.



Kuva 2.5: Asiakas-palvelin malli.

Web-palvelin kuuntelee asiakkaiden HTTP-sivupyyntöjä tietyssä IP-osoitteessa ja TCP-portissa, ja pyynnöstä riippuen joko palauttaa staattisia tiedostoja kuten kuvia, tai sovelluslogiikkaa vaativissa sivupyynnöissä välittää pyynnön sovelluspalvelimelle. Tyypillisesti web-palvelin ja sovelluspalvelin toimivat samalla fyysisellä palvelinlaitteella ja mahdollisesti sama ohjelmisto tekee molemmat tehtävät. Web-palvelimia ovat esimerkiksi Linux-ympäristössa suosittu avoimen lähdekoodin Apache ja Windows-ympäristöissä toimiva Microsoftin Internet Information Services (IIS).

Sovelluspalvelin vastaa käyttäjän sivupyyntöihin ja muodostaa käyttäjän pyytämät HTML-sivut. Sovelluspalvelimia ovat esimerkiksi Java EE määrittelyn mukaiset Oracle WebLogic Server, IBM WebSphere ja JBOSS. Myös muille ohjelmointikielille on olemassa sovelluspalvelimia, kuten Zend Server PHP:lle ja Microsoftin .NET ympäristöt.

Sovelluksen toimintoja suorittaessa voidaan tarvita yhteyksiä muihin järjestelmiin sekä tietokantoihin. Tyypillisesti tällaisia järjestelmiä ovat käyttäjän tunnistamiseen Lightweight Directory Access Protocol (LDAP) -protokollalla käytettävät hakemistojärjestelmät, kuten Microsoftin Active Directory (AD), sekä bisneslogiikkaan liittyvät räätälöidyt taustajärjestelmät. Palvelukeskeisessä arkkitehtuurissa (eng Service Oriented Architecture, SOA) eri taustajärjestelmien toimintoja tarjotaan web-sovellukselle itsenäisinä sovelluspalveluina (eng Web Service, WS) rajapinnan kautta.

2.4 Perinteisen web-sovelluksen edut ja ongelmakohdat

Perinteisen tyyppisiä web-sovelluksia on tehty pitkään, ja siten niihin liittyvät tekniikat, tuotteet, työkalut ja käytännöt ovat kypsiä. Kaupallisia ja avoimia tuotteita on saatavilla laajasti, kuin myös niiden osaajia ja tukea. Perinteisten web-sovellusten yksinkertaiset käyttöliittymät toimivat kaikenlaisissa päätelaitteissa vanhoillakin selaimilla ja ovat siten lähes kaikkien internetin käyttäjien käytettävissä. Koska esityslogiikka on yksinkertaista ja perustuu pääasiassa CSS-tyylimuotoiluun, tuki eri päätelaitteille tarkoittaa käytännössä sivun tyylittelyä niin että se esitetään eri selai-

missa samalla tavalla. Nykyaikaisilla selaimilla se on aiempaa helpompaa. Koska sovelluslogiikka sijaitsee palvelinpuolella, on sovelluksen ajoympäristö hyvin tunnettu ja aina samanlainen. Tämä vähentää toteutuksen monimutkaisuutta. Sivupyyntöjen määrä on kohtuullinen ja pyyntöjä tapahtuu suhteellisen harvoin.

Nykyaikana käyttäjät ovat jo tottuneet modernien internetsivustojen tarjoamaan helppokäyttöiseen käyttöliittymään ja nopeaan vasteaikaan esimerkiksi Facebookin ja Googlen palveluiden kautta. Siten sovelluksiin kohdistuvat vaatimukset ovat kasvaneet. Perinteisen web-sovelluksen avulla tehtävä työ pitää suorittaa sivun kokoisina kokonaisuuksina. Käyttäjälle annettava vaste on hidas ja koostuu kokonaisen sivun noutamisesta oheistiedostoineen palvelimelta, ja sen muodostamisesta selaimessa yhtenä suurena kokonaisuutena. Sivun latautumista odottaessa käyttäjän työvirta katkeaa, koska latauksen aikana ei ole mahdollista olla vuorovaikutuksessa sovelluksen kanssa. Tämä keskittymisen katkeaminen on usein syynä tyytymättömyyteen ja pitkillä vasteajoilla voi se johtaa sovelluksen käytön lopettamiseen. Asiakkaan ja palvelimen välillä liikutetaan paljon turhaan toistuvaa dataa, kun ladataan uusi sivu joka on suurimmalta osin täysin sama kuin edellinen sivu. [48]

Tavanomaiset asiakkaiden päätelaitteet ovat nykyisin tehokkaita. Päätelaitteet tarjoavat sovellukselle hajautetun suoritusympäristön, jonka resurssien käyttö on sovelluksen tarjoajalle maksutonta. Perinteisen web-sovelluksen asiakaslaitteiston resurssit jäävät suurelta osin hyödyntämättä, kun esitys- ja bisneslogiikka suoritetaan palvelimella.

2.5 Yhteenveto

Perinteisen web-sovelluksen tiedonsiirto asiakkaan ja palvelimen välillä tapahtuu synkronisesti sivukokonaisuuksina. Käyttäjä ja palvelin odottavat vuorollaan toisiaan, eikä sovelluksen käyttöliittymä ole käytettävissä silloin kun sivua muodostetaan palvelimella. Tästä aiheutuu hitaat vasteet käyttäjälle ja matala vuorovaikutuksen taso.

Sovelluksen toimintalogiikka sijaitsee sovelluspalvelimella ja siihen liittyvissä taustajärjestelmissä. Sivun sisältö muotoillaan palvelinsovelluksessa HTML-merkintätavalla ja esitetään selaimessa CSS-tyylisääntöjen perusteella. Selaimessa suoritettava toimintalogiikka rajoittuu yksinkertaiseen lomaketietojen oikeellisuuden tarkastukseen. HTML-sivut ja niihin liittyvät resurssit pyydetään palvelimelta HTTP-protokollaa hyödyntäen, TCP-protokollan tarjoamalla yhteydellä. Käyttäjälle näkyvä navigointi sivulta toiselle perustuu URL-osoitteisiin ja domain-nimiin, jotka käännetään DNS-palvelun avulla koneiden kesken käytettäviksi IP-protokollan osoitteiksi.

Perinteisen web-sovelluksen vahvuus on tekniikoiden ja tuotteiden kypsyys. Laadukkaita tuotteita ja niiden osaajia on hyvin saatavilla. Koska sovelluslogiikka rajoittuu palvelinpuolelle, on sovelluksen suoritusympäristö muuttumaton ja hyvin tunnettu. Palvelimella ajettavan sovelluksen tarjoajan on kuitenkin ostettava kaikki sovelluksen vaatima laskentateho, kun toisaalta käyttäjien laitteistot olisivat hajautettu ja edullinen suoritusympäristö sovellukselle. Sovelluksiin kohdistuvat vaa-

timukset ovat muuttuneet käyttäjien totuttua rikkaisiin internetpalveluihin, kuten Facebookin ja Googlen palvelut. Siksi perinteinen web-sovellus tuntuu vanhanaikaiselta ja hankalalta käyttää.

Luku 3

Rikas internetsovellus

Tässä luvussa käsitellään rikkaan vuorovaikutuksen mahdollistavia tekniikoita, joiden avulla on mahdollista tehdä web-sovelluksista työpöytäsovelluksen kaltaisia. Rikkaiden internetsovellusten toteutustekniikat voidaan jakaa Asynchronous Javascript and XML (AJAX) -tekniikkoihin ja liitännäisiin (eng. plug-in) perustuviin sovelluksiin. Tässä luvussa esitellään kumpikin toteutusympäristö ja syvennytään tarkemmin AJAX- ja HTML-tekniikoihin.

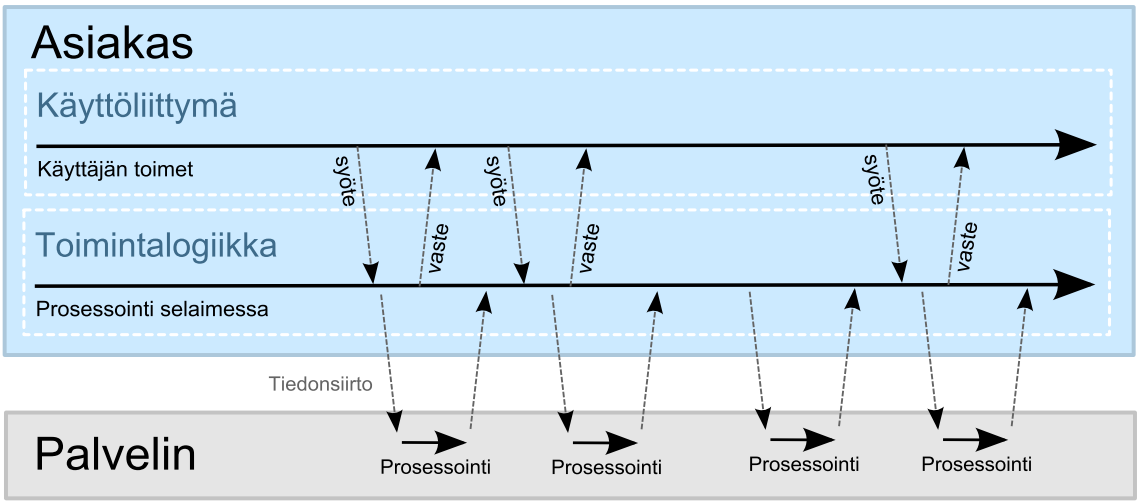
3.1 Rikkaan internetsovelluksen ominaispiirteitä

Web-sovellusten käyttöliittymille asetetut vaatimukset ovat muuttumassa webin käytön yleistyessä ja tekniikoiden kehittyessä. Rikkaat internetsovellukset (eng. Rich Internet Application, RIA) vievät web-sovelluksen käyttöliittymää työpöytäsovellusten suuntaan ja ovat osittain syrjäyttämässä perinteisiä työpöytäsovelluksia ja natiiveja mobiilisovelluksia. Rikkaassa internetsovelluksessa aiempaa suurempi osa sovelluslogiikasta suoritetaan palvelimen sijaan käyttäjän tietokoneella ja tyypillisesti asiakkaan ja palvelimen välinen liikenne on asynkronista. [12]

Rikkaat internetsovellukset auttavat rikkomaan useita perinteisen web-sovelluksen rajoituksia, kuten synkronisen sivunvaihdon ja käyttäjälähtöisen pull-tyyppisen tiedonvaihdon. Sovelluksen käyttöliittymästä voidaan tehdä vuorovaikutteisempi ja graafisesti rikkaampi.

Rikkaille internetsovelluksille ominaista on että sovelluksen tilaa voidaan synkronoida asynkronisesti käyttöliittymän ja palvelimen välillä. Tiedon vaihto tapahtuu käyttäjälle näkymättömästi taustatoimintona, ilman sivunvaihtoa. Käyttäjälle näkyvät sivunlataukset voivat rajoittua yhteen sovelluksen käynnistävään aloitussivun avaamiseen. Kuva 3.1 esittää käyttäjän, selainsovelluksen ja palvelimen välisen asynkronisen kommunikoinnin. Kuvasta on nähtävissä, että käyttäjä voi saada syötteilleen vasteen selainsovelluksen toimintalogiikalta jo ennen kuin syötteen prosessointi palvelimella on valmis. Selainsovelluksen toimintalogiikka voi vaihtaa tietoa palvelimen kanssa omatoimisesti esimerkiksi tietyin aikavälein, ilman käyttäjän syötettä tai ilman näkyvää vastetta käyttäjälle.

Haasteena on esittää käyttäjälle mikä tieto on todella tallennettu käyttöliittymästä palvelimelle asti. Käyttäjä tarvitsee tiedon milloin sovelluksesta poistuminen



Kuva 3.1: Rikkaan internetsovelluksen käyttäjän, selainsovelluksen ja palvelimen välinen asynkroninen kommunikointi. [20]

on turvallista menettämättä tietoja. Koska sovelluksen tila ei perustu sivunvaihtoihin, on myös haasteellisempaa palata tiettyyn sovelluksen tilaan tiettyä osoitetta käyttäen.

Toisaalta taustajärjestelmien aiheuttamat muutokset voidaan tuoda käyttöliittymään ilman käyttäjän toimia, jolloin käyttäjällä käyttöliittymässä oleva tieto voidaan pitää mahdollisimman ajantasaisena ja voidaan välttää suuret konfliktit. Mahdollisuus joustavaan synkronointiin tuo myös uudenlaista monimutkaisuutta sovellukseen, kun käyttöliittymän tilaa synkronoidaan palvelimelle ja samalla reagoidaan palvelimelta tuleviin muutoksiin. Synkronointien hallinnassa auttaa selkeä arkkitehtuuri.

Rikkaat internetsovellukset voidaan jakaa toteutustekniikoiden perusteella Asynchronous Javascript and XML (AJAX) -tekniikkoihin ja liitännäisiin (eng. plug-in) perustuviin sovelluksiin. AJAX-tekniikoilla toteutetut sovellukset toimivat käyttäjän selaimen JavaScript-moottoria ja HTML-tekniikoita hyödyntäen, eivätkä vaadi käyttäjältä nykyaikaisen selaimen lisäksi muita asennuksia. Liitännäisiin perustuvat sovellukset sen sijaan vaativat jonkin sovellusympäristön asentamisen käyttöjärjestelmään tai selaimen liitännäiseksi. [30]

3.2 Liitännäisiin perustuvat tekniikat

Suljettuihin liitännäistekniikoihin perustuvat sovellukset eivät ole täysin riippuvaisia yleisesti standardoitujen tekniikoiden kehityksestä, ja siksi liitännäistekniikat ovat tarjonneet ensimmäisenä edistyneitä tapoja hyödyntää asiakaslaitteiston resursseja, kuten tallennustilaa ja suorituskykyisiä näytönohjaimia. Liitännäistekniikat tarjoavat myös sovelluksen istunnon käsittelyn, joka keventää palvelimen kuormitusta. [30]

Haittapuolena liitännäisiin perustuvat tekniikat edellyttävät käyttäjältä sovellusympäristön asentamista ja ylläpitoa. Lisäksi liitännäisten saatavuus rajoittaa asiakasympäristöjen vaihtoehdot niihin joille liitännäisellä on tuki. Sovelluksen kehittäjiä liitännäisiin perustuvien tekniikoiden käyttö sitoo sovelluskehityksen tietyn yrityksen tarjoamiin ja usein suljettuihin tekniikoihin. [30]

Tähän työhön liittyvän sovelluskehitysprojektin kehittäjät tuntevat HTML- ja JavaScript -tekniikat entuudestaan hyvin. Liitännäisiin perustuvista tekniikoista heillä ei ole kokemusta, ja niiden käyttöönotto edellyttäisi koulutusta. Työhön liittyvä sovellus ei hyödy liitännäistekniikoiden tarjoamista kehittyneistä multimediaominaisuuksista tai monipuolisesta graafisuudesta, eikä sovelluksen käyttäjiltä haluta edellyttää liitännäistekniikoiden asentamista ja ylläpitämistä. Tämän luvun jälkeen työssä keskitytään vain HTML- ja JavaScript-tekniikoihin.

Adobe Flash on joukko multimediatekniikoita, jotka mahdollistavat animaatiot ja vuorovaikutuksen käyttäjän selaimessa Flash Player -liitännäisen avulla. Aluksi Flash oli tarkoitettu graafiseen työhön ja animaatioihin, mutta on sittemmin laajentunut paremmilla sovelluskehitysmahdollisuuksilla ja Adobe Flex -ympäristöllä. Flash tunnetaan parhaiten selaimen liitännäisenä, mutta Flash-sovelluksia on mahdollista suorittaa myös ilman selainta Adobe AIR -sovellusympäristössä. [30]

Adobe on siirtänyt Flex -ympäristön kehitysvastuun avoimen lähdekoodin yhteisölle ja alkanut tukemaan myös HTML-tekniikoiden kehitystä. Flex-ympäristö tukee Flash-tekniikan lisäksi myös sovellusten kääntämistä useille mobiilikäyttöjärjestelmille. HTML5-tekniikoiden suosion kasvun myötä Adobe keskittyy kehittämään Flash-tekniikoita pelaamiseen ja videotoistoon. [1]

Tuki Adobe Flash Player -liitännäiselle on laskussa huippuvuosista. Näkyvimpänä puutteena Flash -tuessa ovat olleet Applen iOS-käyttöjärjestelmää käyttävät laitteet, joissa ei ole koskaan ollut tukea Flashille, mutta joissa HTML5 on kattavasti tuettu. Flash Player -liitännäisen kehitys mobiilialustojen web-selaimille on lopetettu version 11.1 jälkeen [1].

JavaFX on Sun Microsystemsin (nykyisin Oracle) esittelemä ohjelmistoalusta. Ennen versiota 2.0 JavaFX-sovellukset kirjoitettiin JavaFX Script -skriptikielellä. Versiosta 2.0 lähtien JavaFX on toteutettu natiivina Javan kirjastona ja JavaFX-sovellukset kirjoitetaan Java-kielellä. JavaFX Script -kieli on tiukasti yhteydessä Java-kieleen ja hyötyy Javan tehokkaaksi kehittyneestä ajonaikaisesta ympäristöstä (Java Runtime Environment, JRE) ja virtuaalikoneesta (Java Virtual Machine, JVM). Sovelluskehittäjille on tarjolla NetBeans ja Eclipse -kehitysympäristöt ja graafisille suunnittelijoille liitännäiset suosittuihin Adobe Photoshop ja Adobe Illustrator -ohjelmiin. [41]

JavaFX ei rajoitu käyttäjän selaimeen, vaan se voidaan ajaa mobiilisovelluksena matkapuhelimissa tai työpöytäsovelluksena tietokoneella. Kaikkien JavaFX-sovellusten käyttäminen edellyttää, että käyttäjä on asentanut JRE-sovellusympäristön. Oracle on vapauttanut JavaFX -tekniikat avoimen lähdekoodin OpenJDK-projektiksi versiosta 2.0 alkaen [35]. [41]

Silverlight on Microsoftin liitännäiseen perustuva tekniikka, joka on maksuton ja toimii useissa selaimissa, alustoissa ja laitteissa. Silverlight perustuu .NET-sovelluskehykseen ja on siten hyvä vaihtoehto kehittäjille jotka käyttävät Microsoftin .NET-

tekniikoita, C#-ohjelmointikieltä ja Visual Studio -sovelluskehitystyökalua (eng. Integrated development environment, IDE).

Silverlight-liitännäisen suurin ongelma on sen levinneisyys, joka ei ole Flash ja Java -liitännäisten tasolla. Silverlightin kohdalla tuleekin erityisesti arvioida sovelluksen kohderyhmän halukkuus asentaa uusi liitännäinen. [18]

3.3 Asynkroninen JavaScript ja HTML

Kaikki nykyaikaiset selaimet tukevat JavaScript-koodin suorittamista selaimessa. Tämä mahdollistaa interaktiivisten HTML-sivujen luomisen ja esityslogiikan suorittamisen käyttäjän selaimessa. HTML5-tekniikka mahdollistaa aiempaa rikkaamman graafisen toteutuksen sekä paremmat mahdollisuudet hyödyntää asiakaskoneen laitteistoresursseja. Näin on mahdollista luoda liitännäisiin perustuvaa rikasta internetsovellusta vastaava toteutus ilman vaatimusta liitännäisten asennuksesta.

JavaScript-kieli

JavaScript on ilmaisuvoimainen ohjelmointikieli, jolla voidaan laajentaa sovelluslogiikkaa käyttäjän web-selaimeen. Suurin osa nykyaikaisista web-sivustoista käyttää JavaScriptiä ja kaikissa nykyaikaisissa web-selaimissa on tuki JavaScript-koodin tulkitsemiseen. Siksi se on erittäin laajasti levinnyt ohjelmointikieli. [17]

Nimestään huolimatta JavaScript-kielellä on syntaksin ja nimeämiskäytäntöjen lisäksi hyvin vähän yhteistä Javan kanssa. Kieli on dynaamisesti tyypitetty, eli muuttujien arvot ovat tyypitettyjä, mutta muuttujat eivät. JavaScript on luokkamäärittelyiden sijaan prototyyppeihin perustuva olio-ohjelmointikieli, jossa kaikki oliot ovat avain-arvo hakurakenteita (eng. associative array), ja myös funktiot ovat olioita. Funktiot voivat sisältää funktioita ja niitä voidaan käsitellä muuttujina, jotka suoritetaan käyttämällä muuttujan nimessä ()-merkkejä. Mitä tahansa funktiota voi myös käyttää uuden olion luomisessa. `New`-avainsana luo uuden olion prototyyppiin perustuen ja käyttää annettua funktiota olion rakentajana (eng. constructor). [17] [47]

JavaScript-kielen on esitellyt vuonna 1995 silloin johtava selainvalmistaja Netscape. Aluksi sitä käytettiin hyvin yksinkertaiseen lomakkeiden validointiin ja muu sovelluslogiikka sijaitsi tyypillisesti Perl-kielellä kirjoitetuissa palvelinpuolen sovelluksissa. Java-tuotemerkin vuoksi Microsoft on käyttänyt JavaScript-toteutuksissaan virallisesti nimeä JScript, mutta kyseessä on täysin sama kieli. Vuodesta 1997 lähtien JavaScript-kielen ydin on standardoitu ECMAScript-nimellä. [59]

Asynkroninen lataus ja Web 2.0

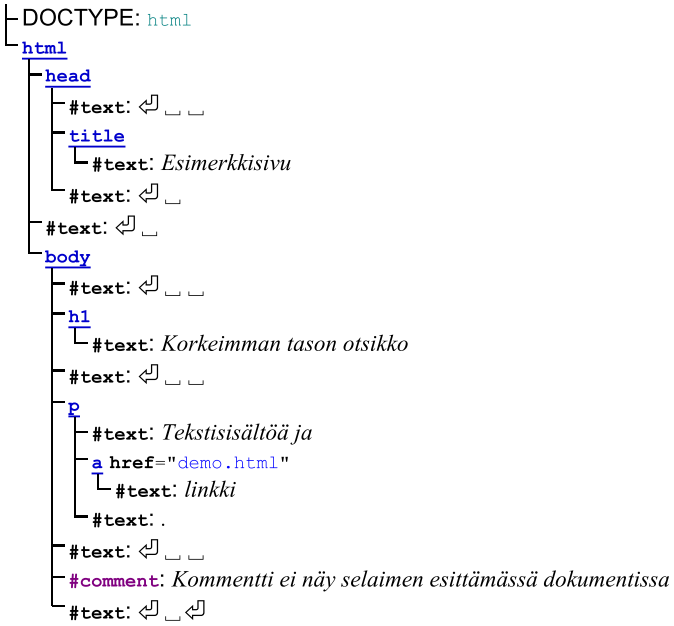
Selainten tarjoama XMLHttpRequest (XHR) -rajapinta mahdollistaa JavaScript-sovelluksen ja palvelimen välisen kommunikaation taustalla ilman että käyttäjän tarvitsee ladata kokonaan uutta sivua. Suosittu termi Asynchronous JavaScript and XML (AJAX) tarkoittaa XHR-rajapinnan hyödyntämistä JavaScript-sovelluksessa. [59]

Kehittäjille, joille HTML ja JavaScript ovat entuudestaan tuttuja, on huomattavasti helpompaa luoda rikas internetsovellus AJAX-tekniikoita käyttäen. AJAX ei edellytä muutosta käytettyyn ohjelmointikieleen tai kehitystyökaluihin. Lisäksi AJAX-tekniikoita on mahdollista ottaa käyttöön perinteisessä web-sivustossa vaiheittain. Tekniikan alkuvaiheen haasteena oli selainyhteensopivuus eri selainvalmistajien versioiden kesken. Eri selainten tukemisessa auttavat useat valmiit JavaScript-sovelluskehykset, jotka pyrkivät tekemään eri selainten tukemisen kehittäjälle näkymättömäksi. AJAX-toimintojen käyttöön on tarjolla useita sovelluskehyksiä ja kirjastoja, ja sopivimman valitseminen tuo omat haasteensa. [23]

Ensimmäiset suositut AJAX-tekniikkaa hyödyntävät Web 2.0 -sovellukset, kuten Google Maps, muuttivat käsityksiä JavaScriptin tarjoamista mahdollisuuksista. Ne osoittivat että JavaScript-kieltä hyödyntämällä voidaan luoda rikkaita websovelluksia, joiden käyttöliittymä vastaa monipuolisuudessaan liitännäisiin perustuvia internetsovelluksia ja jopa perinteisiä työpöytäsovelluksia, mutta toimivat ilman asennuksia vaatien pelkästään nykyaikaisen web-selaimen. [36]

Document Object Model

Document Object Model (DOM) on alusta- ja kieliriippumaton rajapinta XML- ja HTML-dokumenttien käsittelyyn. DOM-rajapinnan kautta voidaan lukea, muokata, poistaa tai lisätä XML- ja HTML-dokumenttien sisältöä. DOM-mallissa dokumentin sisältö esitetään puuhierarkkisessa rakenteessa, kuten kuvassa 3.2. Selainten DOM-rajapinnat on toteutettu JavaScript-kielellä, ja siten kaikkea käyttäjän näkemää HTML-sivun sisältöä voidaan käsitellä JavaScript-sovelluksissa selaimen tarjoaman rajapinnan kautta. [2]



Kuva 3.2: Esimerkkisivun (kuva 2.2) DOM-puu.

DOM-mallin standardoinnista vastaa World Wide Web Consortium (W3C). DOM-mallin tarkoituksena on ollut mahdollistaa selainriippumaton ja yhtenäinen tapa käsitellä HTML-dokumentteja, mutta alkuvaiheen epäselvän ja osittain puutteellisen määrittelyn vuoksi selainten DOM-toteutuksissa on lukuisia eroja. Selainten välisten erojen vuoksi web-sovellusten toteutuksissa on tyypillisesti tarvittu poikkeuskäsittelyitä käyttäjän selaimen perusteella. Eri selainten välisen yhteensopivuuden saavuttamisessa auttaa JavaScript-peruskirjaston, kuten jQuery [52], tarjoamat selainriippumattomat toiminnot. [46]

HTML5

HTML5 on uusin HTML-kielen standardoitu versio, joka tuo useita interaktiota merkittävästi edistäviä ominaisuuksia HTML-kieleen. HTML5-termillä tarkoitetaan usein myös HTML5, CSS3 ja JavaScript -tekniikoiden yhdessä muodostamaa kokonaisuutta. [24]

HTML5 Canvas-rajapinta tarjoaa aiempaa huomattavasti kattavammat mahdollisuudet tehdä graafisia sovelluksia. Canvas mahdollistaa ensimmäistä kertaa alueen vapaalle piirtämiselle ilman liitännäisiä tai HTML-elementtien rajoitteita. Sitä hyödyntämällä voidaan tehdä näyttäviä pelejä ja animaatioita. Canvas-elementtiä on selaimesta ja laitteistosta riippuen myös mahdollista käyttää kolmiulotteisena versiona.

Yksi CSS-tyylimäärittelyiden kolmannen version merkittävimmistä uusista ominaisuuksista on responsiivisuuden mahdollistava Media Query. Responsiivisella sivulla määritellään erilaiset tyylit esimerkiksi käyttäjän päätelaitteen näytön leveyden tai tarkkuuden perusteella. Media Query -määrittelyiden avulla sama HTML-sivu voidaan sopeuttaa erilaisille näytöille, kuten työpöytätietokoneelle, tablet-laitteelle ja matkapuhelimelle, ilman erillisiä HTML-versioita samasta sisällöstä. Merkittäviä uusia CSS-ominaisuuksia ovat myös siirtymät (eng. transition), muunnokset (eng. transform) ja animaatiot, jotka mahdollistavat monipuoliset HTML-elementtien liikkeet tehokkaasti laitteiston näytönohjainta hyödyntäen.

Uusimpien selainten tarjoama Web storage -rajapinta tarjoaa evästeitä (eng cookies) joustavamman tallennustilan käyttäjän tietokoneelle. Tilaa on enemmän ja sen käsittely tapahtuu vain pyydettäessä selaimen ja JavaScript-sovelluksen välillä. Kaikki evästeet siirrettiin aina jokaisen sivupyynnön mukana luoden ylimääräistä kuormaa verkkoliikenteeseen. Web storage koostuu kahdesta osasta: istunnon ajan säilyvästä session storage -tallennustilasta ja pidempiaikaisesta local storage -tallennustilasta. Geolocation-rajapinta tarjoaa käyttäjän sijaintitiedot, mikäli laitteistossa on paikannusmahdollisuus.

3.4 Selainsovelluksen ja palvelimen välinen tietoliikenne

Rikkaassa internetsovelluksessa tietoa siirretään käyttöliittymäsovelluksen ja palvelinsovelluksen välillä ilman käyttäjälle näkyviä sivunvaihtoja. Palvelinrajapinnan

ja selainsovelluksen väliseen tiedonsiirtoon on suunniteltu Representational State Transfer (REST) -toimintamalli. Tietoliikenne ei aina ole asiakkaan käynnistämää (PULL), vaan tietoa voidaan siirtää myös PUSH-tyyppisesti palvelimelta asiakkaalle kun taustajärjestelmissä tapahtuu muutoksia. Tiedon välittämiseen käytetään perinteisen XML-merkintätavan lisäksi yksinkertaisempaa JavaScript Object Notation (JSON) -merkintätapa.

Representational State Transfer

Rikkaat internetsovellukset lähettävät ja vastaanottavat tietoa sivulatausten välillä palvelimelle ja palvelimelta takaisin. Roy Fielding esitteli väitöskirjassaan [14] Representational State Transfer (REST) -toimintamallin. REST on muodostunut käytännössä standardiksi palvelumalliksi asiakkaan ja palvelimen ohjelmointirajapinnan (eng Application Programming Interface, API) välillä moderneissa websovelluksissa. Käytännössä REST ei ole oikea standardi, vaan monin tavoin tulkittu joukko ohjesääntöjä ja rajoitteita. Se käsittelee vain tietorakenteita ja niiden tilan välittämistä. [4]

REST-määrittelyn täyttäviä sovelluksia kutsutaan englanninkielisellä nimellä RESTful. REST on tulkittu usein löyhästi ja nimitystä voi nähdä käytettävän myös sellaisten rajapintojen yhteydessä, jotka eivät aidosti täytä REST-mallin vaatimuksia. Roy Fielding on tarkentanut REST-mallin määritelmää väitöskirjansa julkaisun jälkeen. [15]

REST toimii tyypillisesti HTTP-protokollan päällä. Resursseihin kohdistuvia operaatioita ovat luonti, luku, päivitys ja poisto (eng Create/Read/Update/Delete, CRUD), joita vastaavat HTTP-protokollan komennot POST, GET, PUT ja DELETE. REST-pyynnön vastauksena voi olla HTML, XML tai JSON sen perusteella, mitä mediatyyppejä HTTP-pyynnön Accept -otsikkotiedossa on määritelty. [4]

Resurssit tunnistetaan ja erotetaan toisistaan niiden URL-osoiteiden perusteella, jotka esitetään muodossa:

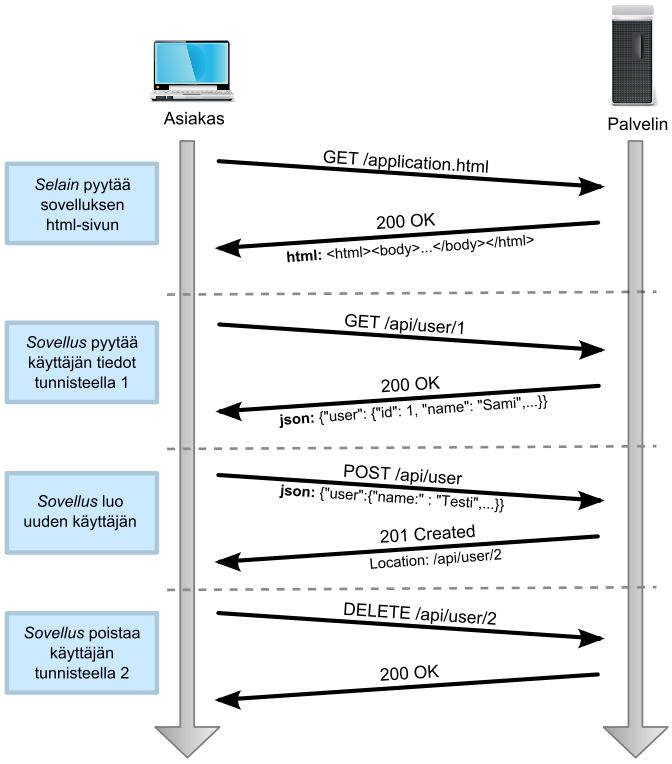
```
Protokolla://Isänta/Sovelluspolku/Resurssityyppi/Resurssitunniste
```

Protokolla, isäntä (eng. host) ja sovelluspolku toimivat nimiavaruutena ja resurssityyppi ja -tunniste ilmentävät tiettyä resurssia. [4]

Kuvassa 3.3 esitetään REST-mallin mukaista verkkoliikennettä asiakkaan ja palvelimen välillä. Aluksi käyttäjän selain pyytää sovelluksen aloitussivun sekä mahdollisesti muita siihen liittyviä tiedostoja. Tämän jälkeen selaimeen ladattu sovellus suorittaa resursseihin operaatioita yksinkertaisina AJAX-kutsuina: ensin haetaan käyttäjän tiedot GET-kutsulla, seuraavaksi luodaan uusi käyttäjä POST-kutsulla ja lopuksi poistetaan luotu käyttäjä DELETE-kutsulla.

REST suunnitteluperiaatteita ovat: [22]

- Tilattomuus: Jokainen asiakkaalta palvelimelle lähtevä pyyntö sisältää riittävästi tietoa pyynnön käsittelemiseksi palvelimella. Pyyntö ei ole riippuvainen palvelimen tietämästä tiedosta.



Kuva 3.3: REST-mallin mukainen verkkoliikenne asiakkaan ja palvelimen välillä.

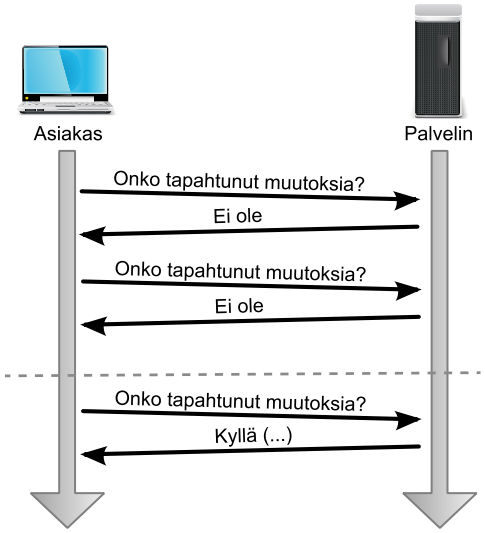
- Yhtenäinen rajapinta, joka koostuu rajatusta operaatioiden ja sisältötyyppien valikoimasta. Esimerkkinä HTTP-operaatiot GET, PUT, POST ja DELETE, ja sisältötyypit text/html ja application/json
- Asiakas-palvelin malli: asiakas pyytää tietoa palvelimelta
- Yksilöllisesti nimetyt resurssit: Unified Resource Identifier (URI)
- URL-osoitteilla toisiinsa yhteydessä olevat resurssien esitykset
- Mahdollisuus tallentaa palvelimen vastauksia välimuistiin verkkoliikenteen tehokkuuden parantamiseksi

PUSH-tyyppinen tietoliikenne

Perinteinen web-sovelluksen asiakas-palvelin liikenne on PULL-tyyppistä, jolloin asiakas pyytää eli ”vetää”palvelimelta tarvittavia resursseja. Kaikki palvelimelta lähtevä liikenne on vastauksia asiakkaan pyyntöihin. PUSH-tyyppisessä liikenteessä asiakas ilmoittaa palvelimelle haluavansa seurata jotakin tietoa, ja palvelin lähettää eli ”työntää”tämän jälkeen viestejä asiakkaalle aina kun tietoon tulee muutoksia, ilman että asiakas erikseen pyytää jokaista palvelimelta lähtevää viestiä. Näin asiakas saa aina ajantasaisen tiedon muutoksista, ihanteellisessa tilanteessa ilman tarpeetonta verkon kuormitusta.

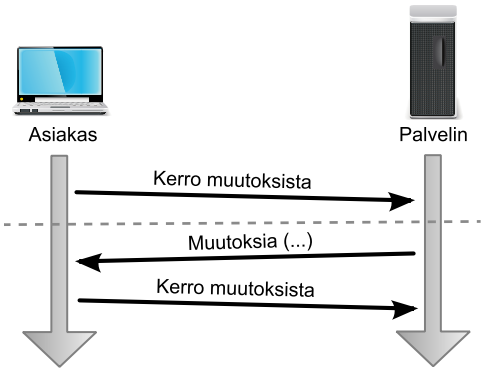
HTTP-protokolla ei tarjoa PUSH-tyyppistä liikennettä palvelimelta asiakkaalle, mutta tätä rajoitusta voidaa kiertää erilaisin tekniikoin. PUSH-tyyppistä liikennettä pystytään jäljittelemään HTTP-protokollaa käyttäen kahdella tavalla: joko palvelimen vastausta pitkään odottamalla (eng long polling) tai moniosaisella palvelinvastauksella (eng streaming). Tekniikoilla on yhteinen nimitys Comet, joka on humoristisesti valittu pesuainebrändi - vastaavasti kuin Ajax. Muita tunnettuja nimiä ovat HTTP server push, Ajax push ja HTTP Streaming.

Jos halutaan seurata jonkin tiedon muuttumista käyttäen PULL-tyyppistä tiedonsiirtoa, joudutaan lähettämään palvelimelle tietyin aikavälein kyselyjä, vaikka tieto ei välttämättä olekaan muuttunut (eng. polling). Tämä lähestymistapa on parhaimmillaan silloin kun tieto muuttuu ennalta arvattavin aikavälein ja voidaan välttää tarpeettomien kyselyiden lähettäminen. Jos ei ennalta tiedetä milloin tieto muuttuu, saadaan suuremmalla kyselytiheydellä nopeammin tieto muutoksista, mutta aiheutetaan myös enemmän tarpeetonta verkkoliikennettä niiden kyselyiden osalta, joiden vastauksena ei ole muutoksia. Kuva 3.4 esittää polling-tyyppisen toistuvan kyselyn asiakkaalta palvelimelle tilanteessa jossa vasta useiden kyselyiden jälkeen palvelimen vastauksena on muutoksia. [17]



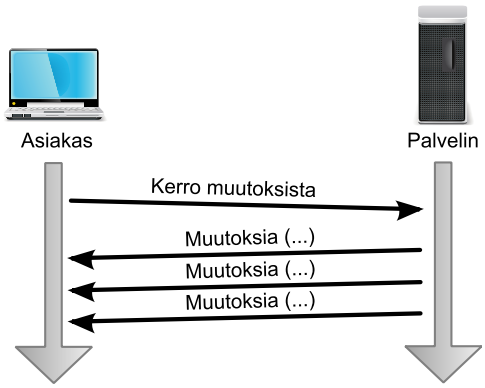
Kuva 3.4: Muutosten tiedustelu toistuvin kyselyin (polling).

Tilanteessa, jossa tieto muuttuu ennalta arvaamattomin aikavälein, voidaan jäädä odottamaan vastausta palvelimelle lähetettyyn kyselyyn siihen asti että vastauksena saadaan todellinen muutos (eng long polling). Palvelin ei vastaa kyselyyn heti, vaan pitää yhteyttä auki ja lähettää vastauksensa vasta sitten kun on jotakin todellista lähetettävää. Aina vastauksen saatuaan asiakas lähettää uuden pyynnön palvelimelle ja jää odottamaan vastausta. Tällöin vältytään tarpeettomalta verkkoliikenteeltä, mutta palvelimelle kertyy paljon avoinna olevia yhteyksiä. Kuva 3.5 esittää long polling -tyyppisen kyselyn asiakkaalta palvelimelle. Kuvasta nähdään, että asiakas aloittaa kysymällä muutoksia, odottaa pitkän ajan ja saa lopulta palvelimelta muutoksia. Muutosten jälkeen avataan uusi odotusjakso kysymällä muutoksia. [17]



Kuva 3.5: Muutosten tiedustelu palvelimen vastausta odottamalla (long polling).

Koska palvelin voi lähettää vastauksensa moniosaisena (käsitellään luvussa 5.1), voidaan samaan pyyntöön kirjoittaa myöhemmin jatkoa pitämällä yhteyttä jatkuvasti auki. Näin asiakas voi saada päivityksiä tilaamaansa tietoon jäämällä odottamaan jatkoa palvelimen lähettämään moniosaiseen vastaukseen. Tätä tekniikkaa voi hyödyntää esimerkiksi iframe-kehykseen jatkuvasti latautuvana sivuna, joka täydentyy hiljalleen ajantasaisilla päivityksillä (eng forever frame, streaming). Tässä tapauksessa asiakkaan ei tarvitse lähettää uutta pyyntöä aina kun palvelimelta saadaan päivitys. Kuva 3.6 esittää asiakkaan lähettämän kyselyn muutoksista, sekä jatkuvat päivitykset palvelimelta muutoksien tapahtuessa, ilman toistuvia kyselyitä asiakkaalta. [17]



Kuva 3.6: Muutosten tiedustelu moniosaisella vastauksella (eng. streaming).

Palvelinlähtöiset tapahtumat (eng. Server-Sent Events) on standardoitu tapa lähettää viestejä palvelimelta selainsovellukseen Comet-periaatteita hyödyntäen. Uusimpien selainten tarjoama EventSource-rajapinta tarjoaa JavaScript-selainsovellukselle valmiin tavan käyttää moniosaisena palvelinvastauksena toteutettua HTTP PUSH-viestintää. Selainsovellus käsittelee viestejä tapahtumina ja selaimen rajapinta vastaa kaikesta tiedonsiirrosta. Palvelimen on lähetettävä moniosainen vastauksensa HTTP-protokollan otsikkotiedolla "`Content-Type: text/event-stream`"va-

rustettuna tekstitiedostona. Tiedostoon kirjoitetut "`data:`"alkuiset rivit toimivat selaimessa tapahtumina muodostettavina viesteinä. [27] [49] [33]

Perinteiset web-palvelimet on suunniteltu avaamaan ja sulkemaan HTTP-yhteyksiä niin nopeasti kuin mahdollista. Koska Comet-tekniikat hyödyntävät pitkään auki pidettäviä yhteyksiä, aiheutuu palvelimelle poikkeuksellisen suuri määrää yhtäaikaisia yhteyksiä. Suosittu Apache-palvelin on suunniteltu käsittelemään noin 10 000 yhtäaikaista yhteyttä, kun Comet-tekniikoita hyödyntävä sovellus voi hyvin vaatia yli 50 000 yhteyttä. Comet-tekniikoita varten on suunniteltu palvelinohjelmistoja jotka suoriutuvat tekniikoiden yhteysvaatimuksista, mutta HTTP-protokollan käyttämisestä aiheutuu edelleen useita TCP-yhteyksiä sekä HTTP-pakettien mukana lähetettävien laajojen otsikkotietojen aiheuttamaa kuormitusta. [17]

Comet-tekniikat ovat parhaimmillaankin HTTP-protokollan rajoitteiden kiertämistä, joten parempi tapa PUSH-tyyppiseen liikenteeseen olisi TCP-yhteyksien hyödyntäminen ilman HTTP-protokollaa. Selaimessa suoritettavalle sovellukselle ei kuitenkaan tietoturvasyistä voida antaa vapaata pääsyä TCP-tason tiedonsiirtoon, vaan sen päälle tarvitaan uusi protokolla huolehtimaan tietoturvasta. WebSocket on HTML5-kokonaisuuden yhteydessä esitelty protokolla ja selaimen ohjelmointirajapinta, joka hyödyntää yhtä TCP-yhteyttä asiakkaan ja palvelimen väliseen jatkuvaan viestintään. WebSocket on suunniteltu mahdollisimman yksinkertaiseksi ja kevyeksi protokollaksi. Se toimii HTTP-protokollalle varatuissa porteissa ja hyödyntää HTTP-protokollan yhteyden muodostamisessa käytettävää kättelyä (eng handshake). WebSocket vaatii palvelinsovellukselta tuen uuden protokollan käyttöön, ja sen käyttöönotto voi siten olla hankalampaa kuin HTTP-protokollaa hyödyntävien palvelinlähtöisten tapahtumien käyttö. [13]

Tiedon esitysmuodot

Extensible Markup Language (XML) ja JavaScript Object Notation (JSON) ovat laajasti käytettyjä tiedon merkintätapoja tiedon vaihtamiseen järjestelmien välillä. Tietoa keskenään vaihtavia järjestelmiä ovat esimerkiksi rikkaan internetsovelluksen sovelluspalvelin ja sille kyselyitä lähettävä käyttöliittymäsovellus käyttäjän selaimessa.

Extensible Markup Language (XML) on HTML-kielen tapaan osajoukko SGML-merkintäkielestä. XML:n tavoitteena on ollut käytettävyys internetissä, yksinkertaisuus, kohtuullinen selkeys ja luettavuus. Toisin kuin HTML-kielessä, XML-kielessä ei ole valmiiksi määriteltyjä tagien nimiä, vaan sallitut tagit on käyttäjän määriteltävissä. XML-tiedostojen MIME-mediatyyppejä ovat `application/xml` ja `text/xml` ja tiedostopääte on `.xml`. Listauksessa 3.1 esitetään esimerkki hierarkkisesta xml-tiedosta. Esimerkissä määritellään käyttäjän nimi ja osoitetiedot. [7]

JavaScript Object Notation (JSON) -merkintätapa on johdannainen JavaScript-kielestä ja siten ECMAScript-standardista. Sen suunnittelutavoitteina on olla tekstimuotoinen, pienikokoinen ja siirrettävä. JSON-merkintätavan avulla voi esittää primitiivityypeistä merkkijonot (string), numerot, totuusarvot (boolean) ja tyhjä (null) sekä jäsennetyistä tyypeistä taulukko (array) ja objekti. JSON-tyyppisen tekstin MIME-mediatyyppi on `application/json` ja tiedostopääte on `.json`. [10]

Listaus 3.1: XML-esimerkki

```
<?xml version="1.0" encoding="UTF-8" ?>
<user id="1">
  <name>Teemu Teekkari</name>
  <address>
    <street>Esimerkkikatu 2</street>
    <areacode>12345</areacode>
    <city>Vantaa</city>
    <country>Finland</country>
  </address>
</user>
```

Listaus 3.2: JSON-esimerkki

```
{
  "user": {
    "id":   1,
    "name": "Teemu Teekkari",
    "address": {
      "street":   "Esimerkkikatu 2",
      "areacode": 12345,
      "city":     "Helsinki",
      "country":  "Finland"
    }
  }
}
```

XML-esimerkkiä 3.1 vastaava tieto on kuvattu esimerkissä 3.2 JSON-muodossa. JSON on sellaisenaan käytettävissä JavaScript-tietorakenteena selaimissa ja siksi se on erityisen tehokas vaihtoehto rikkaan internetsovelluksen käyttöön. XML-merkityn tiedon parsiminen on selaimissa raskaampaa, koska natiivin JavaScript-tietorakenteen sijaan parsittua XML-tietoa käsitellään hierarkkisen DOM-puun kautta. JSON on myös merkintätavaltaan yksinkertaisempi kuin XML ja sen avulla voidaan esittää sama tieto pienemmällä merkkien määrällä. Tutkimusten mukaan JSON on tehokkaampi kuin XML, ja sen käsittely vaatii JavaScript-sovellusympäristössä vähemmän resursseja. [34] [21]

3.5 Selainsovelluksen vaatimukset palvelinpuolelle

Rikkaan internetsovelluksen vaatimukset sivupyyntöihin vastaavalle palvelinsovellukselle ovat erityyppisiä kuin perinteisessä internetsovelluksessa, jossa palvelinsovellus muodostaa valmiita HTML-sivuja sisältäen kaiken sivuun liittyvän sisällön. Myös rikkaan internetsovelluksen tapauksessa sovelluspalvelimen pitää osata muodostaa vähintäänkin html-sivun runko, josta selaimessa toimiva käyttöliittymä voi käynnistyä. Sivun muotoiluun liittyvät vaatimukset ovat kuitenkin usein palvelinpuolella perinteistä web-sovellusta yksinkertaisemmat, koska suuri osa muotoilusta voidaan tehdä selaimessa suoritettavassa käyttöliittymässä.

Uutena vaatimuksena sovelluspalvelimen pitää tarjota rajapinta, jonka avulla käyttöliittymäsovellus saa pyytämänsä tiedon halutussa muodossa. Rajapintaan kohdistuu todennäköisesti paljon enemmän HTTP-pyyntöjä kuin mitä perinteisen web-sovelluksen palvelimelle, joten rajapinnan pitää suoriutua nopeasti ja tehokkaasti suuresta määrästä yksinkertaisia pyyntöjä.

Nykyaikaisista palvelinpuolen sovelluskehyksistä esimerkiksi Java-ympäristössä toimiva Spring, Ruby-kieleen perustuva Ruby On Rails (tunnetaan myös nimellä Rails) ja Groovy-kieleen perustuva Grails (aiemmin tunnettu nimellä Groovy on Rails) tarjoavat työkalut nopeaan ja vaivattomaan REST-palveluiden toteuttamiseen. Tähän työhön liittyvässä sovelluskehitysprojektissa otetaan käyttöön Grails-sovelluskehys, joka tarjoaa nykyaikaisen ympäristön palvelinsovellukselle.

Grails on dynaaminen web-sovelluskehys joka toimii Java-ympäristössä. Se perustuu Groovy-ohjelmointikieleen, joka käännetään Javan tapaan tavukoodiksi Java-virtuaalikoneella ajettavaksi. Siten Grails-sovellukset toimivat samoilla sovelluspalvelimilla kuin muutkin Java-sovellukset. Grails pyrkii käytäntöjen avulla vähäisempään konfiguraatioon ja helpompaan käyttöönottoon. Yhteisen tavukoodin vuoksi Groovy-sovelluskoodista voidaan kutsua Java-kielellä toteutettuja luokkia, ja Grails hyödyntääkin tunnettuja Java-sovelluskehyksiä, kuten Spring ja Hibernate. [11]

3.6 Yhteenveto

Rikkaat internetsovellukset perustuvat asynkroniseen tiedonsiirtoon: tietoa voidaan lähettää ja vastaanottaa käyttäjälle näkymättömästi ilman sivunvaihtoja. Siten on mahdollista käsitellä sivun sisältöä pieninä osina ja antaa käyttäjälle nopeita vasteita sivulle tehtyihin muutoksiin. Liitännäisiin perustuvat tekniikat ovat tuoneet ensimmäisenä korkean graafisuuden ja vuorovaikutuksen web-sovelluksiin, mutta nykyisin vastaava on toteutettavissa HTML-tekniikoita hyödyntäen ilman käyttäjältä vaadittavia liitännäisten asennuksia.

AJAX-tekniikat mahdollistavat asynkronisen HTTP-tiedonsiirron selainsovelluksesta palvelimelle selainrajapintoja hyödyntäen. JavaScript-kielellä voidaan luoda selaimessa suoritettavaa sovelluslogiikkaa, jonka käytettävissä ovat kaikki selainten tarjoamat rajapinnat. DOM-rajapinta mahdollistaa HTML-dokumentin käsittelyn selainsovelluksessa ja sen avulla sivun muodostamisen vastuu voidaan siirtää palveli-

melta selainsovellukselle. HTML5-tekniikat tuovat useita uusia rajapintoja käyttäjän laitteistoresurssien tehokkaampaan hyödyntämiseen.

Rikkaiden internetsovellusten palvelinpuolen sovelluksen vaatimukset muuttuvat, kun aiempaa suurempi osa sovelluslogiikasta voidaan suorittaa selaimessa. Palvelinsovelluksen pitää tarjota selainsovellukselle rajapinta, jonka kautta selainsovellus saa pyydettyä tarvitsemansa datan haluamassaan muodossa. REST on suosittu malli selaimen ja palvelimen väliselle liikenteelle, joka tapahtuu tyypillisesti JSON-muodossa.

Luku 4

JavaScript arkkitehtuuri ja suunnittelumallit

Hyvä arkkitehtuuri selkeyttää sovelluskehitystä ja tekee sovelluksesta monin tavoin laadukkaamman. Tässä luvussa esitetään keinoja paremman rakenteen luomiseksi JavaScript-sovellukseen.

Selaimessa toimivan käyttöliittymäsovelluksen toteuttamiseen on tarjolla lukusia suunnittelumalleja ja avoimen lähdekoodin JavaScript-kirjastoja, joiden avulla voidaan saavuttaa selkeä arkkitehtuuri ja välttää tarpeetonta pyörän uudelleen keksimistä.

4.1 Tarve arkkitehtuurille

JavaScript antaa sovelluskehittäjälle hyvin vapaat kädet. Tämän vapauden myötä kirjoittaa helposti sovelluskoodia, jonka kulku on vaikeasti hahmotettavaa ja jonka toiminta saattaa olla arvaamatonta. JavaScript -sovelluksen kehittämisessä auttaa laadukkaat kirjastot kuten jQuery [52], mutta toisaalta kirjastojen käytön helppous voi johtaa harkitsemattomaan rakenteeseen. Tuloksena voi olla sovellus jossa tehdään paljon irrallisia asioita ilman selkeää rakennetta.

Yksinkertaiset JavaScript-sovellukset käsittelevät tyypillisesti suoraan HTML-sivun DOM-rakennetta ja säilyttävät sovelluksen datan pääasiassa DOM-puussa elementtien attribuutteina, lomaketietoina tai muuna sivun sisältönä. Samaa DOM-puun dataa saatetaan käsitellä useasta eri toiminnosta ilman että mikään sovelluksen osa tuntisi datan tilaa jatkuvasti. Jos sovelluskoodia ei ole ryhmitelty selkeiksi kokonaisuuksiksi, ei kehittäjälle ole selvää mitä koodia suoritettaan milloinkin. Nämä seikat voivat helposti johtaa siihen että toteutuksessa on paljon riippuvuuksia eri toimintojen välillä. Sovelluksen laajentuessa tämä lähestymistapa tekee kokonaisuudesta sotkuisen ja vaikeasti hallittavan. Sovellusta laajennettaessa muutosten vaikutukset ovat vaikeasti hahmotettavissa ja muutosten tekemisestä tulee hidasta ja vaikeaa. Tarvitaan selkeä arkkitehtuuri ja suunnittelumallit, jotta sovelluksen rakenne säilyy selkeänä, helposti ylläpidettävänä ja laajennettavana. [37]

Ohjelmistoarkkitehtuuri toimii tavallaan kääntäjänä sovellukselle asetettujen vaa-

timusten ja sovelluksen toteutuksen välillä. Oikein valittu arkkitehtuuri on sovellusprojektin onnistumisen kannalta kriittinen tekijä monin tavoin. Hyvä arkkitehtuuri auttaa saavuttamaan korkean laadun kohtuullisin kustannuksin sekä auttaa varmistamaan että sovellus täyttää sille asetetut tavoitteet suorituskyvyn, luotettavuuden, siirrettävyyden, skaalautuvuuden ja yhteensopivuuden osalta. [19]

Arkkitehtuurikuvaukset auttavat ymmärtämään monimutkaisiakin järjestelmiä yksinkertaistamalla asioita korkeamman tason tarkasteluun. Samalla arkkitehtuurikuvaus tuo esille suunnitelman korkean tason rajoitukset. Asiat selkeiksi kokonaisuuksiksi eriyttävä arkkitehtuuri mahdollistaa uudelleenkäytön usealla tasolla. Arkkitehtuuri tuo helpotusta myös toteutusvaiheessa, kun arkkitehtuurikuvaus määrittelee korkean tason rakenteet ja sovelluksen osien väliset yhteydet ja rajapinnat. Jatkokehitys helpottuu kun arkkitehtuurissa on huomioitu laajentamisen mahdollisuus ja yksittäisten osien korvaaminen paremmilla tulevaisuudessa. [19]

4.2 Suunnittelumallit

Suunnittelumallit ovat toimivaksi todistettuja lähestymistapoja ohjelmistokehityksen haasteisiin. Koeteltujen suunnittelumallien käytöllä pystytään vähentämään arkkitehtuurin suunnitteluun tarvittavaa työtä ja siten pystytään keskittymään kokonaisratkaisun laatuun. Suunnittelumallien käyttö ohjaa tekemään paremmin jäsenneltyä koodia ja vähentää siten refaktoroinnin tarvetta. [38]

Osmani [38] esittelee seuraavia JavaScript-suunnittelumalleja. Esimerkkeinä käytetään laadukkaan ja suositun jQuery-kirjaston [52] toimintoja.

- **Moduulimalli** jäljittelee luokkien käsitettä ja mahdollistaa julkisten ja yksityisten muuttujien yhdistelmät JavaScript-objekteissa. Muuttujia ei JavaScript-kielessä varsinaisesti pystytä määrittämään julkisiksi tai yksityisiksi, mutta sulkeuman (eng closure) avulla voidaan suojata osa muuttujista globaalista näkyvyydestä. Julkiset muuttujat ja metodit voidaan julkaista palauttamalla julkinen rajapinta, jonka kautta voidaan kutsua vain julkiseksi tarkoitettuja toimintoja.

- **Prototyyppimalli** perustuu JavaScriptin natiiviin tapaan toteuttaa perintä luokkien sijaan prototyypin avulla. Prototyyppi toimii mallina, jonka perusteella objektin eri ilmentymät luodaan. Sen sijaan että JavaScript-sovelluksessa pyrittäisiin jäljittelemään luokkiin perustuvien ohjelmointikielien toimintatapoja, on tehokkaampaa hyödyntää prototyyppien vahvuuksia.

- **Singleton-malli** varmistaa että objektista voidaan luoda vain yksi ilmentymä. Uutta objektin ilmentymää luotaessa pidetään kirjaa luonneista ja tarkistetaan, onko vastaava jo olemassa. Jos objektista on jo ilmentymä, palautetaan olemassa oleva ilmentymä uuden luonnin sijaan.

- **Tarkkailijamallissa** (eng. observer pattern) kohdeobjekti ylläpitää listaa siihen liittyvistä tarkkailijaobjekteista ja ilmoittaa tarkkailijoille kun kohteen tila

muuttuu. Tarkkailijamalli tunnetaan myös julkaise/tilaa (eng. publish/subscribe) -nimellä ja se on JavaScript-kehityksessä todella keskeinen suunnittelumalli, sillä HTML-sivun DOM-elementtien tapahtumien hallinta perustuu tähän malliin. Myös jQuery-kirjaston tapahtumankäsittelyn jQuery.on (tilaa), jQuery.trigger (julkaise) ja jQuery.off (lopeta tilaus) -funktiot perustuvat tähän malliin.

- **Välittäjämalli** (eng. mediator pattern) vähentää tarkkailijamallin osapuolten välisiä riippuvuuksia lisäämällä niiden välille yhteisen välittäjän. Osapuolet eivät koskaan kommunikoi suoraan keskenään, vaan aina välittäjän kautta. Tarkkailija- ja välittäjämalleille yhteistä on julkaisun ja tilauksen perusperiaate. Välittäjämallissa kaikki tilaukset ja julkaisut suoritetaan yhden välittäjän kautta, kun tarkkailijamallissa osapuolet voivat tilata ja julkaista keskenään.

- **Komentomalli** (eng. command pattern) tarjoaa ulkopuolisen objektin jonkin toiminnon kutsumiseen, ja mahdollistaa siten toiminnon parametrisoinnin ja kutsumisen erityttämisen toiminnon toteutuksesta.

- **Julkisivumalli** (eng. facade pattern) tarjoaa yksinkertaistetun julkisen rajapinnan joka piilottaa taustalla olevien toimintojen kompleksisuuden. Esimerkiksi jQuery-kirjaston funktiot jQuery.get, jQuery.post ja jQuery.getJSON ovat yksinkertaistettuja versioita taustalla kaiken XHR-toiminnallisuuden toteuttavasta jQuery.ajax-funktiosta.

- **Tehdasmallissa** (eng. factory pattern) uusia objekteja ei luoda suoraan, vaan ne pyydetään tehdastyyppiseltä objektilta. Tehdasobjekti tarjoaa yleisen rajapinnan, jonka kautta parametrien avulla voidaan pyytää useita erityyppisiä objekteja. Tehdasmalli toimii erityisesti tilanteissa joissa objektin luonti on monimutkaista, tai luotavilla objekteilla on paljon yhteisiä piirteitä.

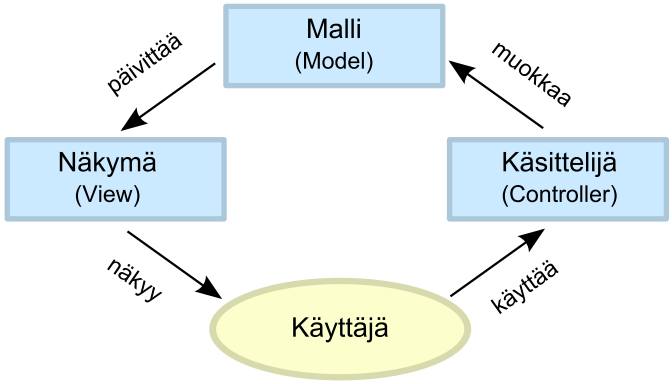
- **Mixin-malli** vähentää toimintojen toistamista laajentamalla prototyyppiä yhteisillä toiminnoilla. Javascriptin saman objektin eri ilmentymät jakavat yhteisen prototyypin, jota laajentamalla voidaan helposti lisätä toiminnallisuus kaikkiin objektin ilmentymiin.

- **Koristelijamalli** (eng. decorator pattern) parantaa koodin uudelleenkäyttöä laajentamalla uutta toiminnallisuutta olemassa oleviin objekteihin. Alaluokkien sijaan objektiin voidaan lisätä useita uusia koristelijaobjekteja progressiivisesti. Näin yhdestä objektista saadaan tehtyä useita variaatioita ilman että alkuperäinen objekti muuttuu. Esimerkki koristelijasta on jQuery.extend-funktio, jonka avulla voidaan yhdistää useita objekteja yhdeksi objektiksi.

- **Koontimallissa** (eng. composite pattern) objektikokonaisuus koostuu useista objekteista, mutta kokonaisuutta käsitellään samalla tavalla kuin yksittäistä objektia. Esimerkiksi koontimallin nimeä muuttamalla voidaan muuttaa kokonaisuuden kaikkien objektien nimet samalla tavalla kuin muutettaisi yksittäisen objektin nimeä. Suositun jQuery kirjaston keskeinen jQuery-objekti

toimii koontimallin mukaisesti aina samalla tavalla, riippumatta siitä sisältääkö se vain yhden vai useampia objekteja.

4.3 Malli, näkymä ja käsittelijä



Kuva 4.1: MVC-mallin osapuolten yhteydet toisiinsa

Model-View-Controller (MVC) -suunnittelumalli erittelee mallin (eng. model), näkymän (eng. view) ja käsittelijän (eng. controller) erillisiksi sovelluksen osiksi. Malli kuvastaa sovelluksen dataa, näkymä käyttöliittymää ja käsittelijä niitä toimintoja jotka reagoivat käyttäjän syötteisiin. Suosittu MVC-arkkitehtuuri on laajasti käytössä palvelinpuolen web-sovelluksissa, joissa myös näkymä muodostetaan pääasiassa palvelinpuolella. Kuvassa 4.1 esitetään MVC-mallin komponentit suhteessa käyttäjään, sekä osapuolten välisen vuorovaikutuksen suunta. [37]

Suosituissa JavaScript-kirjastoissa käytetään usein perinteisen MVC-mallin sijaan MVC-mallin johdannaisia, joita voidaan kutsua yleisnimityksellä MV*. Model-View-Presenter (MVP) -suunnittelumalli korvaa MVC-mallin käsittelijän esittäjällä (eng. presenter) ja Model-View-ViewModel (MVVM) -malli näkymämallilla (eng. viewmodel). Joissakin malleissa käsittelijän paikalla on reitittäjä (eng. router), joka hallitsee sovelluksen URL-osoitteiden käsittelyyn liittyvät toiminnot ja vastaa siitä että sovelluksen URL-osoite ja tila vastaavat toisiaan. [37]

Käyttöliittymäsovelluksen MV*-kirjasto auttaa sovelluksen rakenteen luomisessa, yksinkertaistaa synkronointia palvelinpuolen kanssa, eriyttää DOM-puun sovelluksen datasta ja tarjoaa DOM-puun, mallin ja kokoelmien synkronoinnin.

Malli (eng. model) edustaa sovelluksen dataa. Malli voi olla esimerkiksi puhelinluettelosovelluksen henkilö tai uutissovelluksen uutinen. Se ei ota kantaa käyttöliittymään tai esitystapoihin, vaan ilmoittaa tarkkailijoille kun muutoksia tapahtuu, jotta tarkkailijat voivat reagoida omalta osaltaan datan muutoksiin. Tyypillisesti MV*-kirjasto tarjoaa myös kokoelman (eng. collection), jonka avulla voidaan ryhmitellä mallin ilmentymiä suuremmiksi kokonaisuuksiksi. Palvelinpuolen kanssa yhteiset tietomallit selkeyttävät ja yksinkertaistavat käyttöliittymän ja palvelinsovelluksen välistä viestintää.

Näkymä (eng. view) on visuaalinen esitys sovelluksen mallin tilasta. Tyypillisesti näkymä on sovelluksen käyttöliittymä. Näkymä tarkkailee mallin muutoksia ja päivittää sovelluksen DOM-rakenteita muutosten tapahtuessa. Näkymä voi käyttää sivun muodostamisessa sivupohjia (eng. template), joihin dynaamiset tiedot voidaan syöttää esimerkiksi JSON-muodossa. Sivupohjien käsittelyyn on tarjolla useita erillisiä JavaScript-kirjastoja.

Käsittelijä (eng. controller) on välikäsi mallin ja näkymän välillä. Käsittelijä varmistaa että mallin ja näkymän tilat vastaavat toisiaan: se tekee muutoksia näkymään kun malli muuttuu ja malliin kun käyttäjä tekee muutoksia sovelluksen näkymässä. Joissakin MVC-kirjastoissa käsittelijän ja näkymän raja ei ole näin selvä, vaan käsittelijän toiminnallisuutta saatetaan sisällyttää näkymään.

Esittäjä (eng. presenter) sisältää näkymään liittyvän logiikan ja tekee näkymästä yksinkertaisemman kuin MVC-mallissa. Ero MVC-malliin on semanttinen.

Näkymämalli (eng. viewmodel) toimii tiedon muuntimena. Se muuntaa mallin informaation näkymälle sopivaan muotoon, ja välittää komentoja näkymältä mallille. Näkymämallin tekemä muutos voi olla esimerkiksi päivämäärän muunnos näkymän suomalaisesta formaatista mallin unix-aikaleimamuotoon.

4.4 Modulaarisuus ja vastuiden jako

Moduulien avulla voidaan rakentaa laajoja sovelluksia, jotka toimivat suunnitellulla tavalla myös silloin kun kokonaisuus koostuu erilaisista lähteistä peräisin olevasta koodista. Harkitulla globaalin nimiavaruuden käytöllä sovellus toimii, vaikka kaikkien moduulien olemassaolo ei olisikaan odotettua. [17]

JavaScript-kielessä ei ole valmiiksi selkeää tapaa ohjelmallisesti tuoda (eng. import) ohjelmakoodia riippuvuuksien perusteella. Perinteisesti kaikki HTML-sivulla mahdollisesti jossain vaiheessa tarvittavat skriptitiedostot lisätään sivuun script-elementteinä. Kaikki koodi ladataan heti sivun latauksen yhteydessä riippumatta siitä käytetäänkö osakokonaisuuksia lainkaan. Natiivi ratkaisu tähän puutteeseen on tulossa seuraavien JavaScript-versioiden yhteydessä, mutta modulaarisuus on jo nyt toteutettavissa kirjastojen avulla. [38]

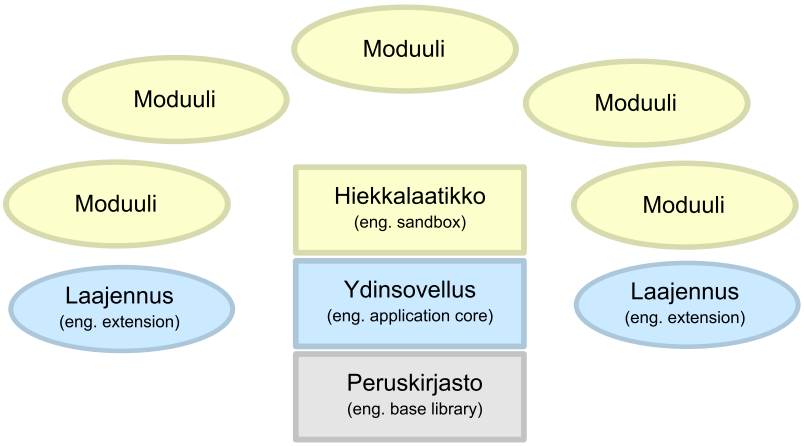
Asynchronous Module Definition (AMD) -rajapinta määrittelee selainsovelluksen moduuleja tavalla jolla sekä moduulit että niiden riippuvuudet voidaan ladata asynkronisesti silloin kun niitä tarvitaan, jos niitä tarvitaan. Asynkronisuus mahdollistaa paremman suorituskyvyn ja käyttökokemuksen, kun aluksi voidaan ladata vain tärkeimmät osat nopeasti ja jatkaa muiden osien lataamista taustalla. Moduulit nimetään merkkijonoilla ja niiden varsinaiset tiedostopolut määritellään erikseen. Siten moduulin toteutus on helppo vaihtaa koskematta moduulikutsuihin muualla sovelluksessa. Moduulit on suljettu erillisiksi nimiavaruuksiksi, eivätkä ne siten saastuta koko sovelluksen nimiavaruutta. AMD-mallin kaksi keskeistä toimintoa ovat moduulien määrittelyyn käytettävä define ja riippuvuuksien lataamiseen käytettävä require. Tuki AMD-rajapinnalle löytyy esimerkiksi suositusta jQuery-kirjastosta. AMD-rajapinta ei määrittele tai toteuta varsinaista moduulien latausta. Moduulien latauksen toteuttavia kirjastoja on useita, tunnetuimpina RequireJS

ja curl. [45]

Selainympäristöön suunnattu AMD on irtautunut CommonJS-projektista. CommonJS on palvelinpuolen JavaScript-sovelluksiin suunnattu tapa määritellä moduuleja, eikä kaikkia sen toiminnallisuuksia ole mahdollista toteuttaa selainympäristössä.

Moduuleilla voi olla irtonainen tai tiivis yhteys toisiinsa. Irtonaisesti kytketyillä (eng. loosely coupled) moduuleilla on hyvin vähän tai ei lainkaan tietoa toisistaan. Tiiviisti kytketyt (eng. tightly coupled) moduulit vastaavasti tuntevat toisensa ja siten koodimuutokset yhteen moduuliin edellyttää myös muiden moduulien muokkausta. Sovelluksen ylläpidettävyys parantuu kun moduulien välillä on mahdollisimman vähän suoria riippuvuuksia, eli kun ne ovat irtonaisesti kytketty. [58]

Nicholas Zakas [58] esittelee vastuualueita osittavan (eng. separation of concerns) moduulimallin, joka koostuu peruskirjastosta, sen päälle rakennetusta sovelluksen ytimestä ja ytimen laajennuksista, ytimeen liitetystä moduuleille yhteisestä "hiekkalaatikosta"sekä lopulta useista moduuleista. Zakasin malli esitetään kuvassa 4.2.



Kuva 4.2: Nicholas Zakasin [58] esittämä JavaScript-sovelluksen moduulimalli

Peruskirjasto, kuten jQuery, tarjoaa selainten välisen normalisoinnin ja yleisesti käytettäviä tai monimutkaisuutta yksinkertaistavia toimintoja. Vain peruskirjasto tietää mitä selainta käytetään.

Ydinsovellus järjestää mallien välisen viestinnän ja mallien elinkaaren hallinnan, virheiden käsittelyn ja laajennusten liittämisen. Se on ainoa sovelluksen osa, joka on suoraan yhteydessä peruskirjastoon. Vastaavasti hiekkalaatikko on ainoa ydinsovellukseen yhteydessä oleva sovelluksen osa. On toivottavaa että ydinsovellus on globaalin nimiavaruuden ainoa olio.

Hiekkalaatikko toimii moduulien näkymänä muuhun sovellukseen ja mahdollistaa moduulien irtonaisen kytkennän. Hiekkalaatikko on rajapinta moduulien ja ydinsovelluksen välillä, eikä se toteuta sovelluksen varsinaista toiminnallisuutta. Kun moduuli haluaa viestiä oman toiminta-alueensa ulkopuolelle, se pyytää luvan ja viestii hiekkalaatikon kautta.

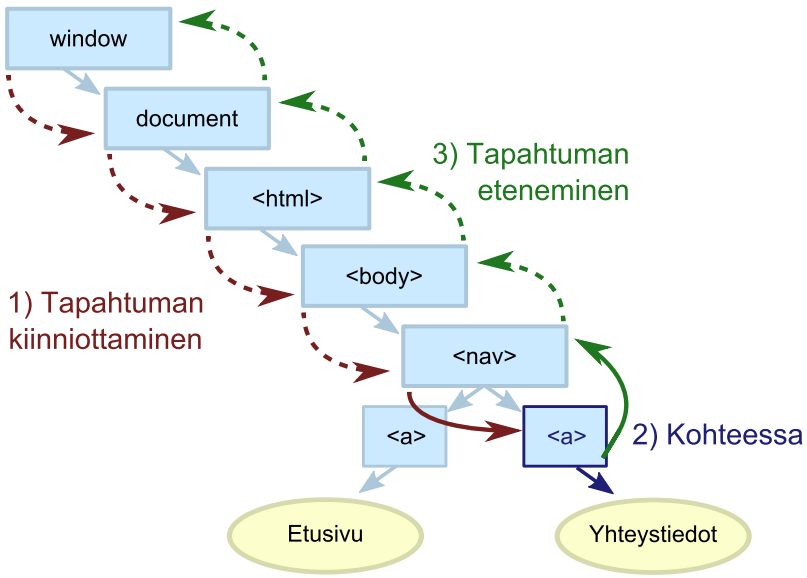
Moduulit ovat itsenäisiä toiminnallisuuden osia, jotka sisältävät bisneslogiikkaa ja osan käyttöliittymästä. Moduuli on yksi sovelluksen käyttöliittymän osa, joka

rakentuu HTML-merkinnästä, CSS-tyylimäärittelyistä ja JavaScript-koodista.

4.5 Tapahtumien ohjaama arkkitehtuuri

JavaScript-selainsovelluksen tapahtumien ohjaama arkkitehtuuri (eng. event-driven architecture, EDA) perustuu DOM-rajapinnan tapahtumien liipaisuun ja kuunteluun. Kun dokumentin elementtiin kohdistuu muutos, siitä luodaan tietyn tyyppinen tapahtuma. Kyseiseen dokumentin elementtiin sidottu tapahtuman kuuntelija saa tiedon seuraamastaan tapahtumasta ja käynnistää tapahtuman käsittelyn. Tapahtumien ohjaama arkkitehtuuri mahdollistaa erittäin irtonaisen kytkennän, kun tapahtuman aiheuttava toiminto ei tiedä mitkä toiminnot reagoivat tapahtumaan tai millaisia toimintoja tapahtuman johdosta suoritetaan.

Tapahtumien eteneminen DOM-puussa koostuu kolmesta eri vaiheesta, jotka esitetään kuvassa 4.3. Tapahtuman kiinniottaminen (eng. event capturing) on ensimmäinen vaihe, jota seuraavat kohteen saavuttaminen ja tapahtuman eteneminen (eng. event bubbling). Kun johonkin DOM-puun elementtiin kohdistuu tapahtuman aiheuttava toimenpide, DOM-puussa edetään juuresta kohti tapahtuman kohdetta sen kiinniottamiseksi. Tässä vaiheessa eteneminen on mahdollista keskeyttää ennen kuin se tavoittaa kohdetta. Kun tapahtuman kohde-elementti on tavoitettu, palataan DOM-haaraa pitkin ilmoittaen kunkin elementin kohdalla tapahtuman etenemisestä. Tiettyyn elementtiin kohdistuvaa tapahtumaa on siten mahdollista seurata paitsi kohde-elementtiin sidotuilla tapahtuman kuuntelijoilla, myös kaikilla niihin elementteihin sidotuilla kuuntelijoilla, jotka sijaitsevat matkalla dom-puun juuresta kohde-elementtiin. Tämä mahdollistaa tapahtumien käsittelyn suurempina kokonaisuuksina, jolloin voidaan esimerkiksi seurata kaikkien navigaation linkkien klikkauk-



Kuva 4.3: Tapahtuman eri vaiheet DOM-puussa [40]

sia seuraamalla koko navigaation sisältävän ylemmän tason elementin tapahtumia. Tapahtuman käsittelijä tietää mikä on tapahtuman kohde-elementti, ja missä elementissä tapahtuman eteneminen parhaillaan tapahtuu. Tapahtuman käsittelijän on mahdollista keskeyttää tapahtuman eteneminen, ja siten estää tapahtuman leviäminen DOM-hierarkiassa korkeamman tason elementtien tietoisuuteen. [40] [2]

DOM-puun tapahtumille on ennalta määrättyjä tapahtumanimikkeitä hiiren liikkeille, DOM-puun rakenteen muutoksille ja HTML-elementtien muutoksille. Niiden avulla on mahdollista seurata esimerkiksi klikkauksia (`click`), lomaketietojen muutoksia (`change`), sivun vierittämistä (`scroll`) tai sivun latautumisen valmistumista (`load`). DOM-tapahtumamallissa määriteltyjen tapahtumien lisäksi on mahdollista käyttää sovelluksen omia tapahtumanimikkeitä. Tapahtumien hallintaa yksinkertaistaa JavaScript-peruskirjastot, jotka tarjoavat toteuttajalle selainriippumattoman tavan käsitellä tapahtumia. [40] [2]

4.6 Tietoturvan vaikutus arkkitehtuuriin

Yksi JavaScript-tietoturvan osa-alue on käyttäjän tietoturva ja käyttäjälle arvaamattoman sovelluskoodin ajaminen käyttäjän selaimessa. Toisaalta JavaScript-sovelluksen lähdekoodi ja data on käyttäjän nähtävissä ja muokattavissa, joten sovelluksen kannalta on erityisen tärkeää varmistaa salaisen tiedon ja bisneslogiikan pysyminen salaisena.

Käyttäjän suojaamiseksi selainten ympäristöt rajaavat JavaScript-sovellukset ikkunakohtaiseen hiekkalaatikkoon (eng. sandbox), jonka ulkopuolelle selainsovelluksella ei ole pääsyä. JavaScript-sovelluksen ei siis ole mahdollista saada suoraa pääsyä asiakaskoneen käyttöjärjestelmän resursseihin, kuten tiedostojärjestelmään tai verkkoyhteyksiin. Käyttöjärjestelmän resurssien käyttö tapahtuu selaimen tarjoamien rajoitettujen rajapintojen kautta. Selainrajapinnat tarjoavat esimerkiksi rajoitetut HTTP-yhteydet (XMLHttpRequest-rajapinta), sijaintitiedot (Geolocation-rajapinta) ja tallennustilaa (WebStorage-rajapinta). Yksi tärkeimmistä tietoturvaperiaatteista selainympäristössä on saman alkuperän politiikka (eng. same-origin policy), joka mahdollistaa vapaan vuorovaikutuksen saman sivuston eri sivujen sovellusten kesken, mutta estää pääsyn eri sivustojen resursseihin. [60]

Kun osa sovelluksen toimintalogiikasta viedään käyttäjän selaimeen, syntyy myös huoli siitä voidaanko selaimessa toimivaan sovellukseen ja sen käyttäjään luottaa. Toisin kuin palvelinpuolen sovelluksessa, jonka lähdekoodi on salaista ja täysin palveluntarjoajan hallinnassa, selainsovelluksen tulkittava JavaScript-lähdekoodi on käyttäjän nähtävissä ja muokattavissa. Vihamielinen käyttäjä voi ohittaa sovelluksen toimintoja, kuten syötteiden tarkistuksia, tai laajentaa sovellusta omalla koodillaan. Esimerkiksi kauppasovelluksessa voitaisiin helposti muokata hintoja tai poistaa tilaukseen liittyviä rajoituksia ja tarkastuksia, jos sovellus luottaisi käyttäjään kaiken toimintalogiikan osalta. Samoin kaikki data jota palvelin lähettää selainsovellukselle, on uteliaan käyttäjän saatavilla.

Selainsovelluksessa tehtävä syötteiden validointi mahdollistaa viiveettömän reagoinnin virhetilanteisiin ja siten sujuvan käyttökokemuksen sovelluksen käyttäjälle.

Tarkistuksia ei kuitenkaan voida tehdä ainoastaan selaimessa, vaan käyttäjän syötteiden oikeellisuus tulisi aina varmistaa myös palvelimella. [48]

Osaa bisneslogiikasta ja datan käsittelystä ei voida tehdä selaimessa siitä syystä, että logiikka tai data sisältää liikesalaisuuksia. Tällöin tarvitaan taustajärjestelmien tukea ja osittain valmiiksi prosessoitua dataa käyttöliittymässä esitettäväksi. JavaScript-koodin minimointi tekee koodista vaikeasti luettavaa ja se poistaa myös informaatiota lähdekoodista. Minimoinnilla voidaan poistaa sovelluksen toimintaa selittävät kommentit ja muuttaa muuttujien ja funktioiden nimet merkityksettömään muotoon. Näin voidaan välttää salaisen termistön vuotaminen käyttäjälle, mutta sovelluslogiikka on edelleen osaavan käyttäjän selvitettävissä ja muokattavissa.

Esimerkiksi myyntijärjestelmässä salaista tietoa voi olla hinnan tarkka muodostuminen eri komponenteista, kuten sisäänostohinnasta, katteista ja alennuksista. Silloin hintoja ei voida laskea käyttäjän valintojen perusteella selainsovelluksessa, vaan lopullinen hinta pitää kysyä taustajärjestelmältä. Lisäksi tilausta hyväksyttäessä on taustajärjestelmissä uudelleen varmistettava että myydään oikealla hinnalla, eikä voida luottaa käyttöliittymäsovellukselta saatuun hintatietoon. Vastaavasti saatavuustiedon tarkat komponentit voivat olla salaisia, mutta käyttäjälle halutaan kertoa jotain suuntaa antavaa tietoa saatavuudesta. Useille verkkokaupoille on tyypillistä että ei haluta myydä sellaista tuotetta mitä ei ole enää saatavilla, mutta ei myöskään haluta paljastaa tarkkoja tietoja saatavuuden lisätiedoista kuten tarkasta varaustilanteesta ja myyntimääristä.

4.7 Kirjastot ja sovelluskehykset

Laadukkaita JavaScript-kirjastoja on laajasti saatavilla ja valtaosa niistä on avointa lähdekoodia. MV*-arkkitehtuurin mahdollistavia hyvin tunnettuja kirjastoja ovat Backbone.js, Knockout ja Spine. Laajemmin sovelluksen arkkitehtuuria määritteleviä suosittuja MV*-sovelluskehyksiä ovat Ember.js ja AngularJS. Lisäksi saatavilla on useita muita vastaavia, ja tarjonta kasvaa jatkuvasti.

- **Backbone.js** on suosittu, tehokas ja kevyt kirjasto, joka tarjoaa REST-synkronoinnin palvelinpuolelle, mallit, käsittelijätoiminnallisuuden sisältävät näkymät, tapahtumien ohjaaman toiminnan, sivupohjat ja sivujen reitityksen. Kirjaston on kehittänyt Jeremy Ashkenas. Backbone toimii matalalla tasolla ja jättää paljon tapauskohtaisesti toteutettavaksi. Lisätoiminnallisuutta on saatavilla lukuisten laajennusten avulla. MarionetteJS, Chaplin ja Aura ovat Backbone-kirjaston laajennuksia, jotka helpottavat laajojen JavaScript-sovellusten toteuttamista ja tuovat tarkemmin määrätyn arkkitehtuurin sovellukselle. Backbone-kirjastoa on käytetty useissa suurissa projekteissa, kuten LinkedIn, Hulu, Foursquare, Disqus ja Pandora. Backbonen käyttöön on paljon oppaita ja sen taustalla on aktiivinen yhteisö.

- **Knockout**-kirjasto tarjoaa kaksisuuntaisen sitomisen käyttöliittymän ja tietomallien välillä ja mahdollistaa siten monimutkaistenkin dynaamisten käyttöliittymien sitomisen taustalla oleviin tietomalleihin suhteellisen vaivattomasti.

Se sisältää myös riippuvuuksien seurannan ja siten käyttöliittymäelementit ja niitä vastaavat mallit saadaan pysymään aina toisiaan vastaavassa tilassa. Knockout ei muuten ota kantaa sovelluksen arkkitehtuuriin, ja sitä voidaan käyttää lukuisten muiden sovelluskehysten tai olemassa olevien sovellusten kanssa. Knockout perustuu MVVM-malliin.

- **Spine** on Alex MacCawn kehittämä erittäin kevyt kirjasto, joka tarjoaa yksinkertaisen mallin, näkymän, käsittelijän ja reitittäjän. Spine ei sisällä riippuvuuksia muihin kirjastoihin ja se on joustavasti käytettävissä erilaisiin tarpeisiin. Spinen tavoitteena on yksinkertaisuus ja sen tarkoituksena on jättää paljon vapauksia käyttäjälle.

- **Ember.js** on Yehuda Katzin sovelluskehys, joka tarjoaa enemmän valmiita toiminnallisuuksia ja määrittelee sovelluksen arkkitehtuurin tarkemmin kuin esimerkiksi Backbone. Siinä missä Backbone jättää moduulien väliset yhteydet ja sovelluksen tilan hallinnan käyttäjän toteutettavaksi, tekee Ember.js vastaavia toimintoja automaattisesti. Myös Ember.js kirjastoa on käytetty suurissa ja tunnetuissa sovelluksissa kuten Groupon.

- **AngularJS** tähtää tulevaisuuden selaintoiminnallisuuden, kuten Web Components, tarjoamiseen jo nyt. Angular mahdollistaa HTML-syntaksin laajentamisen sovelluksen tarpeiden mukaisesti. Kirjaston logiikka perustuu toimintaohjeisiin, jotka upotetaan HTML-merkintään laajennetuin elementein ja nykyisten HTML-elementtien ng-alkuisin attribuutein. Kun selain on ladannut HTML-sivun sivupohjaksi, Angular kirjoittaa sivun sisällön uudelleen JavaScript-mallin mukaisesti. Angular on Googlen kehittämä kirjasto. Google käyttää sitä omissa projekteissaan, kuten PlayStation 3 -pelikonsolin YouTube-sovelluksessa.

Useat MVC-kirjastot käyttävät DOM-puun muokkaamiseen sivupohjia (eng template), mutta käyttävät tyypillisesti tähän toiminnallisuuteen erillistä kirjastoa. Suosittuja sivupohjan toteuttavia kirjastoja ovat Handlebars.js, Mustache.js, Underscore.js ja Hogan.js. Ne tarjoavat mahdollisuuden kääntää yksinkertaista logiikkaa ja muuttujia sisältävistä sivupohjista annetun muuttuvan datan sisältäviä HTML-esityksiä. Listauksessa 4.1 esitetään yksinkertainen puhelinluettelosovelluksen sivupohjan syntaksi Handlebars-kirjaston merkintätavalla. Avainsanalla `each` voidaan iteroida läpi `contacts`-listan henkilömallit, jotka sisältävät henkilön etunimen, sukunimen, puhelinnumeron ja sähköpostiosoitteen. Listaus 4.2 esittää käännetyn sivupohjan lopputuloksen kun `contacts`-lista sisältää kaksi henkilöä.

MVC-kirjaston lisäksi erillinen peruskirjasto on usein hyödyllinen. Peruskirjasto tarjoaa yhtenäisen selainrajapintojen käsittelyn niin että sovelluksen korkeampien kerrosten ei tarvitse tietää käytettyä selainta tai tuntea selainten välisiä eroja. Suosittuja peruskirjaston toiminnot sisältäviä kirjastoja ovat jQuery, Prototype, MooTools sekä Dojo ja YUI -kirjastojen ydinkirjastot. Peruskirjaston lisäksi saatetaan tarvita yleiskäyttöiset käyttöliittymäelementit tarjoava kirjasto, kuten jQuery UI, Twitter Bootstrap, Dojo tai YUI-käyttöliittymäkirjasto. Hyödyllisiä yleiskäyttöisiä työkaluja tarjoavat apukirjastot kuten Underscore.js.

Listaus 4.1: Handlebars-sivupohjan syntaksi puhelinluettelosovelluksessa

```
<dl id="phoneBookListing">
  {{#each contacts}}
  <dt>{{this.firstName}} {{this.lastName}}</dt>
  <dd>
    {{this.phoneNumber}}<br />
    {{this.emailAddress}}
  </dd>
  {{/each}}
</dl>
```

Listaus 4.2: Listauksen 4.1 sivupohja käännettynä ja kahden henkilön tiedoilla täytettynä.

```
<dl id="phoneBookListing">
  <dt>Sami Tiilikainen</dt>
  <dd>
    +123 456 789<br />
    sami.tiilikainen@aalto.fi
  </dd>
  <dt>Jukka Manner</dt>
  <dd>
    +123 456 789<br />
    jukka.manner@aalto.fi
  </dd>
</dl>
```

- **jQuery** on erittäin suosittu JavaScript-kirjasto. Sen on alun perin kirjoittanut John Resig ja se on julkaistu vuonna 2006. Kirjastolle ominaista on DOM-elementtien käsittely jQuery-objekteina sekä DOM-puun läpikäynti CSS-valitsimien tapaisesti Sizzle-syntaksilla. JQuery-toiminnot palauttavat yleensä jQuery-objektin ja ovat siten ketjutettavissa. Kirjasto tarjoaa myös kattavan tapahtumien käsittelyn. JQuery on laajennettavissa lukuisilla yksittäisiä toimintoja tarjoavilla laajennuksilla (eng. plugin), sekä käyttöliittymäkomponentit sisältävällä jQuery UI -kirjastolla. [52]

- **Prototype**-kirjasto tarjoaa toimintoja DOM-elementtien valintaan id-tunnisteella tai CSS-syntaksilla, sekä yleiskäyttöisen Ajax-objektin AJAX-kutsujen selainriippumattomaan käsittelyyn. Sen on luonut Sam Stephenson vuonna 2005. Prototype-kirjaston ongelmana on sen perusperiaate laajentaa DOM-rajapinnan ominaisuuksia, jota nykyisin pidetään yleisesti huonona käytäntönä selainyhteensopivuuden ja suorituskyvyn kannalta [46].

- **Dojo Toolkit** on laaja sovelluskehys web-sovellusten kehittämiseen. Se tarjoaa peruskirjaston lisäksi laajoja toiminnallisuuksia ja apuvälineitä MVC-arkkitehtuuriin, lomakkeiden käsittelyyn, käyttöliittymiin, grafiikkaan, mobiilisovelluksiin ja sovelluskehitykseen liittyen. Dojo on useiden nimekkäiden yritysten tukema ja integroitavissa useisiin sovelluskehitysympäristöihin.

- **MooTools** koostuu useista komponenteista, joita voi ottaa käyttöön tarpeen mukaan. Se tarjoaa toiminnot esimerkiksi tyylien, DOM-elementtien, JavaScript-objektien ja AJAX-kutsujen käsittelyyn. MooTools tuo prototyyppeihin perustuvaan JavaScript-kieleen luokkapohjaisen olioiden luonnin `class`-oliota käyttäen. MooTools-kirjaston on julkaissut Valerio Proietti vuonna 2006.

- **YUI** on laaja, useista komponenteista koostuva kirjasto: ydinkomponentti DOM-puun käsittelyyn ja tapahtumien käsittelyyn, aputyökalut lukuisiin perustoimintoihin, käyttöliittymäkontrollit, CSS-komponentit, kehittäjätyökalut testaukseen sekä sovelluskokonaisuuden tiivistämiseen ja dokumentointiin liittyvät työkalut.

- **Underscore** on laadukas kokoelma erilaisia tehtäviä suorittavia toimintoja. Se tarjoaa JavaScript-kieleen selainriippumattomat versiot kokoelmien läpikäyntiin, suodatukseen, hakuun, ryhmittelyyn, yhdistämiseen ja järjestämiseen; monipuoliset tietorakenteet sekä toimintojen toiston rajaamiseen liittyviä toimintoja.

4.8 Yhteenveto

Onnistunut arkkitehtuuri auttaa saavuttamaan korkean laadun kohtuullisin kustannuksin sekä auttaa varmistamaan että sovellus täyttää sille asetetut tavoitteet suorituskyvyn, luotettavuuden, siirrettävyyden, skaalautuvuuden ja yhteensopivuuden osalta. Selainsovelluksen kehityksessä arkkitehtuuri on samalla tavalla tärkeää kuin perinteisen palvelinpuolen sovelluksen kehityksessä. Arkkitehtuuria ei kuitenkaan tarvitse kehittää täysin tyhjästä, vaan voidaan hyödyntää toimivaksi koeteltuja suunnittelumalleja ja sovelluskirjastoja. Palvelinpuolen sovelluksissa laajasti yleistynyt MVC-arkkitehtuuri on selkeä tapa eriyttää malli, näkymä ja käsittelijä erillisiksi kokonaisuuksiksi, myös selainsovelluksissa.

Sovelluskomponenttien välisten riippuvuuksien vähentäminen auttaa parantamaan sovelluksen vakautta ja ylläpidettävyyttä. Irtonaisesti kytketty modulaarinen rakenne mahdollistaa riippuvuuksien poiston. JavaScript ei tarjoa luonnostaan tapaa liittää ohjelmakoodia moduuleittain tarpeen mukaan, mutta tämä on mahdollista saavuttaa kirjastojen avulla.

Kaikkea bisneslogiikkaa ei voida suorittaa selainsovelluksessa, koska lähdekoodi ja muuttujien arvot ovat osaavan käyttäjän nähtävissä ja muutettavissa. Tästä syystä selainsovelluksen välittämiin syötteisiin ei voida luottaa, vaan niiden oikeellisuus pitää varmistaa palvelinpuolen sovelluksessa. Osa toiminnoista täytyy edelleen suorittaa kokonaan palvelinpuolella, jos toimintojen logiikka on salaista tietoa.

Luku 5

Suorituskyky

Tässä luvussa käsitellään rikkaan internetsovelluksen suorituskykyyn vaikuttavia tekijöitä. Käyttäjän kokemaan suorituskykyyn vaikuttaa aiempaa enemmän sovelluksen ulkoasun ja käyttöliittymän sovelluslogiikan käsittely käyttäjän selaimessa, eikä suurimpana pullonkaulana välttämättä ole palvelinsovelluksen tehokkuus tai verkkoliikenteen nopeus.

5.1 Sivun ja siihen liittyvien resurssien lataus

Käyttäjän aistimaan sivun latauksen nopeuteen vaikuttavat useat tekijät. Sivun ja sen resurssien latauksen kokonaisaika koostuu kolmesta päätekijästä: palvelimelle lähetettävästä palvelinpyynnöstä, vastauksen prosessoinnista palvelimella sekä vastauksen ja siihen liittyvien resurssien tiedonsiirrosta.

Verkkoliikenne ja palvelinpyynnöt

Käyttöliittymältään yksinkertaisessa perinteisessä web-sovelluksessa palvelinpuolen suorituskyky on merkittävässä roolissa käyttäjän kokeman suorituskyvyn muodostumisessa. Käyttäjän vuorovaikutuksen viive koostuu suuressa osin sivun vaihtoon liittyvästä verkkoliikenteen viiveestä ja palvelinprosessoinnin kestosta. Tällöin palvelinpuolen suorituskyvyn parantaminen näkyy suoraan käyttäjälle nopeampana sovelluksena.

Rikkaassa internetsovelluksessa käyttäjän kokema viive koostuu useammasta tekijästä ja palvelinpuolen merkitys vähenee, kun osa prosessoinnista siirtyy käyttäjän selaimeen. Tavanomaisesti yli 80 % ajasta käytetään selaimessa HTML-tiedoston latauksen jälkeen, jolloin verkkoliikenteen ja palvelinpuolen alle 20 % osuuden hienosäätö ei ole kaikista tärkein osuus kokonaisuudessa. Käyttöliittymässä käytetystä ajasta suuri osa kuluu sivuun liittyvien staattisten tiedostojen, kuten skriptitiedostojen ja tyylimäärittelyjen hakemiseen. [50]

Riippumatta siitä onko kyseessä rikas vai perinteinen internetsovellus, verkkoyhteyden nopeus on merkittävä tekijä suorituskyvyssä. Erityisesti vuorovaikutteisissa sovelluksissa verkon viive on tärkeä tekijä, mutta tyypillisesti web-sovellukselle

oleellisinta on verkon läpimeno (eng. throughput). Välimuistien (eng. cache) hyödyntäminen auttaa poistamaan tarpeetonta toistoa tiedonsiirrosta. Muuttumattomat tiedostot voidaan lukea käyttäjän välimuistista, jolloin vältytään täysin tiedonsiirrolta käyttäjän ja palvelimen välillä. Välimuistin käytössä auttaa kun skriptit ja tyylimäärittelyt irrotetaan html-sivusta omaan tiedostoonsa, jolloin jokaisen sivun yhteydessä niitä ei tarvitse ladata sivun mukana, vaan ne voidaan lukea erikseen välimuistista. Selaimelle voidaan kertoa tiedoston vanhenemisajankohta HTTP-protokollan Expires-otsikkotiedossa.

Sivun lataukseen liittyvien palvelinpyyntöjen määrää voidaan vähentää yhdistämällä skripti- ja tyylitiedostoja suuremmiksi kokonaisuuksiksi useiden pienempien tiedostojen sijaan. Myös sivuston grafiikoita voidaan yhdistää sprite-kuviksi, jolloin yhteen kuvatiedostoon sisällytetään useita kuvia ja sitä käytetään eri tarkoituksiin näyttämällä vain tietty osa kuvasta. Esimerkiksi Google-haun käyttöliittymän kaikki kuvat ovat yhdessä suuressa sprite-kuvassa. Yksi suuri sprite ei graafisimpien sovellusten tapauksessa välttämättä ole järkevä tapa, koska se paisuisi valtavan suureksi. Mahdollisimman pienien tiedostojen saavuttamiseksi on hyödyllistä ryhmitellä samaan kuvaan saman sävyiset grafiikat, jolloin kuvan väripalettia voidaan rajata mahdollisimman tehokkaasti ja siten saavuttaa pienempi tiedostokoko. Kuvassa 5.1 esitetään jQuery-kirjaston yhden väriteeman ikonit yhtenä Sprite-kuvana.

Tekstisisällöt voidaan pakata, sillä useimmat selaimet osaavat automaattisesti purkaa palvelimen lähettämät GNU zip (gzip) -pakatut http-viestit. Selain, joka osaa purkaa Gzip-pakatun sisällön, lähettää HTTP-pyynnön otsikkotiedoissa `Accept-Encoding` -tietona gzip. Tekstitiedostojen pakkauksella voidaan saavuttaa tyypillisesti 70 % pienempiä tiedostoja. Tyypillisiä tiedostomuotoja joita kannattaa pakata ovat HTML, JavaScript, CSS, XML ja JSON. Kuvatiedostojen pakkaus haaskaa tarpeettomasti suoritinresursseja, koska ne ovat jo valmiiksi pakattuja eikä gzip merkittävästi vaikuta niiden tiedostokokoon.

Kuva 5.1: jQuery UI JavaScript-kirjaston käyttämä, kaikki käyttöliittymäikonit sisältävä sprite-kuva.

Listaus 5.1: Ote jQuery-kirjaston minimoidusta lähdekoodista

```
function cu(a){if(!cj[a]){var b=c.body,d=f("<"+a+">").
   appendTo(b),e=d.css("display");d.remove();if(e==="none
   "||e===""){ck||(ck=c.createElement("iframe"),ck.
   frameBorder=ck.width=ck.height=0),b.appendChild(ck);if
   (!cl||!ck.createElement)cl=(ck.contentWindow||ck.
   contentDocument).document,cl.write((f.support.boxModel
   ?"<!doctype html>":"")+"<html><body>"),cl.close();
```

JavaScript-tiedostojen kokoa voidaan kutistaa pakkauksen lisäksi tiivistämällä (eng minify). JavaScript-koodista voidaan poistaa tarpeettomat osat, kuten kommentit, sisennykset ja rivinvaihdot sekä muuttaa muuttujien nimet ja muu syntaksi mahdollisimman yksinkertaiseen muotoon. Listauksessa 5.1 esitetään ote jQuery-kirjaston minimoidusta lähdekoodista, joka on kokonaisuudessaan tiedostokooltaan alle 40 % normaalista lähdekoodista. Html- ja CSS-dokumenttien tiedostokokoa voidaan rajoittaa pitämällä rakenne mahdollisimman yksinkertaisena ja karsimalla tarpeettomat elementit ja määrittelyt pois.

Rikkaan internetsovelluksen verkkoliikennettä voidaan vähentää lähettämällä AJAX-pyynnöissä ja vastauksissa vain sovelluksen kannalta oleellista tietoa. JSON -tiedonsiirtomuoto on erityisen tiivistä ja sen avulla saavutetaan pienempiä tiedostoja XML-muotoon verrattuna. Hyöty kasvaa entisestään kun verrataan pelkän datan siirtämistä JSON-muodossa valmiiksi muodostetun HTML-sisällön siirtämiseen. Lisäksi myös AJAX-kutsuihin voidaan käyttää välimuistia, jolloin muuttumatonta tietoa ei tarvitse hakea palvelimelta uudelleen.

Jokainen eri domain-osoitteeseen tehty HTTP-pyyntö aiheuttaa DNS-kutsun, ellei domain-nimeä ole valmiiksi selvitetty paikalliseen DNS-muistiin. DNS-kutsut voivat kestää 20-100 millisekuntia, joten eri domain-osoitteisiin tehtävien kutsujen määrän kannattaa kiinnittää huomiota. Toisaalta HTTP/1.1-määrittelyn mukaan selaimen ei tulisi ladata useampaa kuin kahta tiedostoa samalta palvelimelta yhtäaikaisesti, jolloin useampi rinnakkainen lataus voidaan mahdollistaa eri domainnimien käytöllä. Uusimmat selaimet eivät kuitenkaan tarkasti noudata tätä suositusta, vaan lataavat useampia tiedostoja rinnakkain. Vanhemmat selaimet noudattavat tätä rajoitusta, mutta niiden osalta se voidaan osittain kiertää käyttämällä vanhempaa HTTP/1.0 -protokollaa, jossa vastaavaa kahden rinnakkaisen latauksen rajaa ei ole määritelty. [57] [51]

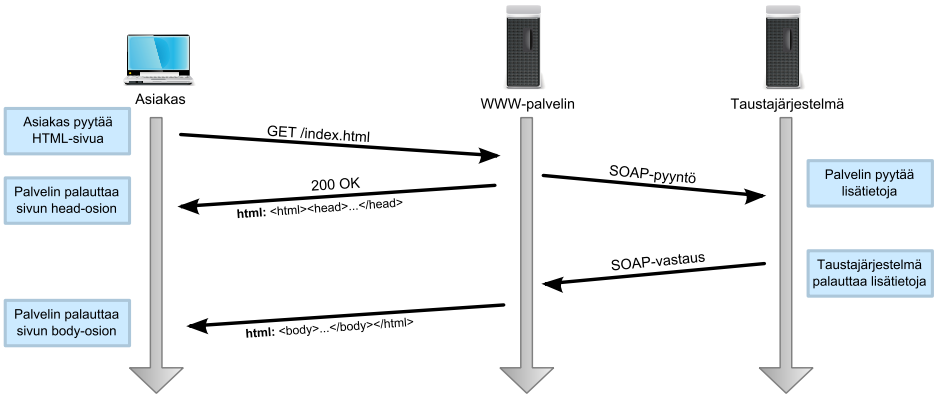
Eri domain-osoitteisiin liittyy myös jokaisella sivupyynnöllä lähetettävät evästeet. Evästeitä on tarpeetonta lähettää sivuun liitettyjä kuva- ja skriptitiedostoja ladattaessa. Erityisesti useita pieniä tiedostoja ladattaessa evästeet http-pyynnössä voivat olla merkittävän suuri osa tiedonsiirrossa. Tarpeettoman evästeiden lähettämisen voi kiertää käyttämällä eri domain-osoitteita tiedostoille jotka käyttävät evästeitä ja tiedostoille jotka eivät käytä evästeitä. [57] [51]

Sivupyynnön käsittely palvelimilla

Joskus sivun muodostaminen vaatii raskasta prosessointia sovelluspalvelimella ja taustajärjestelmissä, jolloin sivupyynnön käsittely palvelimella muodostuu merkittäväksi tekijäksi kokonaisviiveessä. Tätä viivettä voidaan parantaa tehostamalla palvelinpuolen prosesseja ja hankkimalla tehokkaampaa laitteistoa, mutta suorituskykyparannuksia voidaan saada aikaiseksi myös jaksottamalla sivun muodostusta viisaammin.

HTML-sivun toimitus palvelimelta käyttäjälle koostuu useista paketeista. Tyypillisesti sivu muodostetaan palvelimella kokonaisuudessaan, ja vasta sitten lähetetään useana pakettina verkkoon. Raskasta sivua muodostaessa usein sivun head-osio voidaan muodostaa ennen varsinaisen sisällön muodostamista, ja se kannattaa lähettää käyttäjälle jo ennen kuin loppu sivusta on muodostettu. Tällöin selain voi alkaa lataamaan sivun head-osiossa kutsuttavia tyylimäärittelyjä ja JavaScript-tiedostoja samalla kun palvelin muodostaa sivun muuta sisältöä. [16]

Sivun alkuosan lähettäminen muusta sivusta erillään on mahdollista vain HTTP 1.1 -protokollalla, jossa on mahdollista käyttää `Transfer-Encoding: chunked` -otsikkotietoja. Kuvassa 5.2 esitetään osissa lähetettävä vastaus HTTP-pyyntöön. Kuvassa esitetään tilanne, jossa käyttäjä saa lähettämäänsä GET-pyyntöön nopeasti OK-vastauksen sekä HTML-dokumentin head-osion. Vasta taustajärjestelmien prosessoinnin jälkeen palvelin lähettää vastauksen jatkona HTML-dokumentin body-osion. Vanhempaa HTTP 1.0 -protokollaa käytettäessä palvelimelta lähtevässä vastauksessa täytyy olla määritelty koko dokumentin koko `Content-Length` -otsikkotietona, joka käytännössä estää vastauksen lähettämisen ennen kuin sivu on kokonaisuudessaan muodostettu. [16]



Kuva 5.2: HTTP-vastauksen lähettäminen osissa `Transfer-Encoding: chunked` -otsikkotietoja käyttäen.

5.2 Sivun muodostus selaimessa

Sovelluksen esityslogiikan muuttuessa aiempaa monimutkaisemmaksi, kasvavat myös asiakaslaitteiston suorituskykyvaatimukset. Toisin kuin palvelinpuolella, käyttäjien

laitteistoissa on valtavan paljon erilaisia yhdistelmiä laitteistoa ja sovelluksia, jolloin yhtenevän käyttökokemuksen tarjoaminen on haasteellista.

CSS-tyylimäärittelyjen käsittely tapahtuu selaimessa. Käsittelyn tehostamiseksi sivulle kannattaa ladata vain sellaisia määrittelyjä jotka ovat kyseisellä sivulla tarpeellisia. Tästä syystä kaikkien sovelluksen tyylien tiivistäminen yhteen suureen tyylitiedostoon ei ole hyvä idea, jos sovelluksen eri sivut käyttävät eri osia tyylimäärittelyistä.

Selaimet lukevat valitsinsääntöjä oikealta vasemmalle, aloittaen säännön viimeisestä valitsimesta ja käymällä läpi kaikki säännön valitsimet kunnes löytyy osuma tai säännön voi hylätä sopimattomana. Siten erityisesti säännön loppuosalla on suuri merkitys. Mahdollisimman yksinkertainen ja uniikki valitsin on yleensä tehokkain. Jotta elementin valinta olisi yksinkertaista eikä säännön muodostamisessa tarvittaisi monitasoista tagien nimiin perustuvaa hierarkiaa, HTML-dokumentissa kannattaa käyttää elementtejä yksilöiviä id- ja class-attribuutteja. Universaali *-valitsin on raskain, ja myös HTML-tagien nimiin perustuvat valitsimet ovat raskaita.

Sivun muodostuksen kannalta on oleellista missä kohtaa sivun rakennetta tyylitiedostot liitetään sivuun. Selaimet pyrkivät viivästämään sivun muodostamista siihen asti että kaikki tyylitiedostot on ladattu. Tyylit tulisi liittää aina ennen JavaScript-koodia sivun head-osioon. Tällä tavoin saavutetaan tyylitiedostojen mahdollisimman varhainen ja rinnakkainen käsittely, joka mahdollistaa sivun progressiivisen piirtämisen selaimessa samalla kun varsinainen sisältöosuus latautuu. Sivun loppuun upotetut tyylit voivat selaimesta riippuen joko estää sivun progressiivisen muodostamisen, tai aiheuttaa sivun uudelleen piirtoja ja hidastaa siten sivun latautumista. Vastaavasti sivun kuville tulisi määrittää koko jo sivun tyylimäärittelyissä tai sisällön html-attribuutein, jotta kunkin kuvan vaatima tila voidaan päätellä jo ennen varsinaisen kuvatiedoston latautumista. Myöhäisessä vaiheessa tapahtuva kuvien latautuminen pakottaa selaimen piirtämään sivun uudelleen, jos kuva ei asetu sille tyylein varattuun tilaan. [50]

Sivun ulkopuolisten JavaScript-tiedostojen sijainti sivussa on myös erittäin keskeistä sivun muodostamisen kannalta. JavaScript-tiedostojen sopivalla kutsumisella on vielä suurempi vaikutus kuin CSS-tiedostoilla, koska JavaScript-tiedostoa käsiteltäessä selain ei aloita lataamaan muita tiedostoja rinnakkaisesti. Selain ei piirrä progressiivisesti skriptien jälkeistä sisältöä ennen kuin skriptitiedoston käsittely on valmistunut, koska selain ei voi etukäteen tietää lisäävätkö skriptit dokumenttiin uutta sisältöä. Siten sivun alkuun liitetty skriptitiedosto voi merkittävästi viivästyttää koko sivun muodostamista. Vastaavasti aivan sivun loppuun sijoitettu skriptitiedosto ladataan vasta kun muu sisältö on ladattu ja piirretty selaimessa, jolloin käyttäjän kokema kokonaisviive on huomattavasti lyhyempi. [50]

CSS 3 tarjoaa käyttäjän laitteiston suorituskyvyn arvioimiseen näytön resoluution ja pikselitiheyden Media Query -ominaisuuksien avulla. Sen avulla voidaan käyttää kevyempää tyylitystä mobiililaitteille. Yleisen suorituskyvyn mittarina pikselitiheys ja näytön resoluutio on kuitenkin huono, koska sillä ei ole varsinaista yhteyttä laitteiston tehokkuuteen sivun esittämisessä: nykyaikainen mobiililaite voi olla tehokas, ja toisaalta vanha työpöytätietokone voi olla tehoton. CSS-tyyleistä erityisen raskaita ovat `transform` -animaatiot, `border-radius` -pyöristykset ja `box-shadow`

-varjot.

5.3 JavaScript suorituskyky

JavaScript-sovelluksen suorituskykyyn vaikuttaa asiakaslaitteiston suorituskyvyn lisäksi selainten erilaiset tavat suorittaa JavaScript-koodia. Eri laitteistoilla ja eri selaimilla saadaan samalle sovellukselle hyvin erilaisia suorituskykytuloksia. Kaikissa ympäristöissä sovelluksen tehokkuutta parantaa JavaScript-koodin optimointi ja harkittu selaimen tarjoamien rajapintojen käyttö.

Selainten JavaScript-moottorit

Kun JavaScript-sovellukset laajenivat hienostuneiksi ja monimutkaisiksi työpöytäsovelluksen kilpailijoiksi, kohdistui koodia suorittaviin selaimien JavaScript-moottoreihin uudenlaisia suorituskykyvaatimuksia. JavaScript-koodin suoritusnopeudesta tuli ratkaiseva tekijä selaimen käyttökokemuksessa. JavaScript-moottorien kehittyessä huimaa vauhtia aiempaa suorituskykyisemmäksi, tulee JavaScript ohjelmointikielenä aiempaa houkuttelevammaksi ja kypsemmäksi. [29]

Koska dynaamisesti tyypitetyssä JavaScript-kielessä muuttujien tyyppi ja olioiden sisältö voi suorituksen aikana muuttua, se ei ole yhtä tehokkaasti tulkittavissa tai käännettävissä kuin staattisesti tyypitetyt ja luokkamääriteltyihin olioihin perustuvat kielet. [31]

JavaScript ei aina ole ainoa vaihtoehto toiminnallisuuden toteuttamiseksi selainympäristössä, ja vaihtoehtoisia tapoja kannattaa punnita varsinkin raskaissa toiminnoissa. Animaatioita toteutettaessa kannattaakin huomioida mahdollisuus animaatioiden tekemiseen selaimessa CSS-tyylimäärittelyillä JavaScript-koodin sijaan. Koska selaimen CSS-käsittely on kirjoitettu C++ -kielellä ja käännetty valmiiksi ympäristön natiiviksi ohjelmakoodiksi, on sen suoritus väistämättä tehokkaampaa. Siddharth Rao vertaili jQuery-kirjastolla ja CSS-tyylimäärittelyillä tehtyjen animaatioiden suorituseroja käyttäen JavaScript-suoritusnopeudessa tehokkaana tunnettua Google Chrome -selainta. JQueryllä toteutettu JavaScript-animaatio käytti 2119 toimenpidettä ja 6 Mt muistia kun selaimen CSS-käsittelijä suoriutui vastaavasta animoinnista 100 toimenpiteellä ja 1.5 Mt muistimäärällä. [43]

Selainvalmistajien välinen kilpailu on keskittynyt vahvasti JavaScript-koodin suorituksen nopeuttamiseen. Tästä seurannut nopea kehitys on tuonut suuria suorituskykyparannuksia, erityisesti Just-In-Time (JIT) toimintaperiaatteen vuoksi. JIT-kääntäjä (eng Just-In-Time Compiler, JITC) kääntää ajonaikaisesti tulkittavan JavaScript-koodin konekielelle, jolloin JavaScript-sovellus voidaan suorittaa tehokkaasti laitteiston natiivissa muodossa. Kaikki suurimmat selainvalmistajat käyttävät JIT-kääntäjää selainmoottoreissan: Firefoxissa Mozillan TraceMonkey, Chromessa Googlen V8 ja Safarissa WebKit-moottorin SquirrelFish Extreme. Suurin osa mobiiliselaimista käyttää WebKit-moottoria ja sen SquirrelFish Extreme kääntäjää. [31]

Suorituskykyä mittaavien benchmark-testien merkitys on korostunut, koska niiden tulokset ovat konkreettinen vertailukohta ja markkinoitava etu selainten välillä. On kuitenkin yleisesti tiedostettu, myös suorituskykytestien kehittäjien keskuudessa, että testeillä on rajoituksensa eivätkä ne täysin kuvasta selainten välistä suorituskykyeroa todellisten web-sovellusten käytössä. [44]

JavaScript soveltuu yksinkertaisen web-sivuston käyttöliittymälogiikan lisäksi myös monimutkaisempiin sovelluskohteisiin. JavaScript-kielellä on toteutettu esimerkiksi selaimessa täysin ilman liitännäisiä toimiva PDF-lukija PDF.js, sekä MP3-dekooderi JSMad. JavaScript-kielellä yhdessä HTML5 Canvas -rajapinnan kanssa on tehty myös näyttäviä pelejä ja animaatioita.

JavaScript-moottorien kehittyessä suorituskykyisiksi on JavaScript-kielelle löytynyt uusia sovelluskohteita web-selainten ulkopuolelta. Esimerkkinä tästä on Googlen V8-moottoria hyödyntävä Node.js palvelin, joka mahdollistaa palvelinsovelluksen kirjoittamisen JavaScript-kielellä. Node.js palvelin perustuu asynkroniseen palvelinpyyntöjen käsittelyyn, joka hyödyntää säikeistyksen sijaan tapahtumia (eng. event) ja niiden käynnistämiä vastakutsutoimintoja (eng. callback). Selaimessa suoritettavan käyttöliittymälogiikan lisäksi JavaScript-kielellä on siis mahdollista tehdä myös palvelinpuolen sovelluksia ja siten kattaa yhdellä ohjelmointikielellä koko web-sovellus taustajärjestelmistä käyttöliittymään. [53]

JavaScript-optimoinnin suositukset

Vaikka nykyaikaiset JIT-toimintaperiaatteeseen perustuvat JavaScript-moottorit tuovat suorituskykyyn todella merkittäviä parannuksia, joissain tapauksissa optimointi suositeltuja toimintatapoja noudattaen voi auttaa saavuttamaan vielä suurempia suorituskykyparannuksia. JavaScript-moottorit eivät siis edelleenkään vastaa kaikesta koodin optimoinnista. Useimmissa tapauksissa saavutettu optimoinnin hyöty kasvaa entisestään JIT-kääntäjällä verrattuna perinteiseen JavaScript-tulkkiin. [26]

Paikallisten muuttujien käyttö on suositeltavaa paitsi selkeyden, myös suorituskyvyn vuoksi. JavaScript-moottorit nopeuttavat paikallisten muuttujien hakua verrattuna globaaleihin muuttujiin. Poikkeuksena tähän sääntöön on muuttumaton globaalisti käytettävä tieto. Tieto kannattaa asettaa yhteiseen globaaliin muuttujaan kerran sen sijaan että samaa tietoa asetettaisi useassa paikassa eri muuttujiin, koska muuttujan asettaminen on raskaampaa kuin muuttujan arvon hakeminen. Muuttujan uudelleen asettaminen voi kuitenkin olla tehokkaampaa jos olion muuttujan arvoa haetaan toistuvasti, koska on tehokkaampaa hakea arvo paikallisesta muuttujasta sen sijaan että se haettaisi toistuvasti olioon viitaten. Vastaavasti paikallinen muuttuja on tehokkaampi kuin with-lauseen käyttö. Toistuvasti suoritettavan logiikan tulokset on paitsi selkeyden, myös suorituskyvyn kannalta aina järkevää sijoittaa paikalliseen muuttujaan sen sijaan että samaan arvoon päätyvää logiikkaa suoritettaisi toistuvasti peräkkäin. [26] [55]

JavaScript-kielen oliot on tehokkainta luoda JSON-notaatiota käyttäen. Kun objektit luodaan olio-ohjelmoinnin tapaan new-avainsanaa käyttäen, aiheutuu objektin luonnista ylimääräinen funktiokutsu JSON-notaatioon verrattuna. Funktiokutsut ovat JavaScript-kielessä suhteellisen raskaita, koska aina funktiota kutsuttaessa

sen parametreille pitää varata muistia ja asettaa parametrin arvot, sekä etsiä itse funktio sen nimen perusteella. Myös iteraatiossa kannattaa välttää tarpeettomia funktiokutsuja tapauksissa joissa ne ovat vältettävissä eikä niistä erityisesti saada mitään hyötyä. [26] [55]

Eval-funktiolla on mahdollista suorittaa merkkijono JavaScript-koodina. Eval-funktiolle syötettävä teksti pitää parsia ja suorittaa erikseen, ja siten sovelluksen suorituskyky kärsii aina kun tullaan suorituksessa eval-funktion kohdalle. Sen käyttöä tulisi välttää aina kun mahdollista. [26] [55]

DOM-kutsut

DOM-puun elementtejä voidaan hakea, muokata ja luoda selaimen DOM-rajapinnan tarjoamilla toiminnoilla, joiden toteutuksissa on eroja eri selainten välillä. Tyypillisimmät toiminnot ovat elementtien haku id-attribuutilla (`getElementById`), tagin nimellä (`getElementsByTagName`), elementeillä käytetyn luokan nimellä (`getElementsByClassName`) tai css-valitsimella (`querySelectorAll`). Eri selainten toteutusten erojen vuoksi on usein järkevää hyödyntää jotakin peruskirjastoa, kuten jQueryä, DOM-kyselyiden yhtenäistämiseksi.

DOM-rajapinnan kutsut ovat raskaampia kuin muu JavaScript-koodi. Tästä syystä usein käytetyt DOM-puun elementit tai niiden tarvittavat arvot kannattaa sijoittaa JavaScript-muuttujaan sen sijaan että toistuvasti kysyttäisi samaa asiaa DOM-rajapinnalta. Sama pätee myös perustoiminnoissa auttavia kirjastoja kuten jQueryä käytettäessä: JQuery-valitsimien avulla haetut jQuery-objektit kannattaa tallentaa muuttujaan, tai vastaavasti ketjuttaa operaatiot jQueryn tarjoamaa ketjutusta hyödyntäen niin että varsinaiset DOM-pyynnöt jäävät mahdollisimman vähäiseksi. [55]

Jotkin DOM-operaatiot käynnistävät sivun uudelleen piirtämisen, joka kestää aikansa ja hidastaa siten sivun käyttöä. DOM-puuhun muutoksia tehtäessä ne kannattaa tehdä siten, että muutoksesta aiheutuvien uudelleenpiirtojen määrä pysyy mahdollisimman vähäisenä. Esimerkiksi sen sijaan että muokattavien elementtien tyyleistä erikseen muutettaisi useita piirteitä kunkin elementin kohdalla, on tehokkaampaa lisätä elementeille kaikki halutut tyylit sisältävä CSS-luokka. Siten usean uudelleenpiirron sijaan riittää yksi piirto kaikille muutoksille. Myös jotkin lukuoperaatiot, kuten elementtien mittojen kysyminen, saattavat aiheuttaa sivun näkymättömiä uudelleenpiirtoja selaimessa. Toistuvilta tarpeettomilta piirroilta vältytään asettamalla mittasuhteet paikalliseen muuttujaan. [25] [55]

Uusia DOM-elementtejä luotaessa elementit yksi kerrallaan lisättäessä aiheutetaan yhtä monta piirtoa kuin on lisättäviä elementtejä. Sen sijaan voitaisiin irrottaa korkeamman tason elementti DOM-puusta, tehdä siihen tarvittavat lisäykset ja sen jälkeen lisätä kaikki yhtenä DocumentFragment-kokonaisuutena takaisin DOM-puuhun. Tässä tapauksessa aiheutuu vain kaksi uudelleenpiirtoa, yksi elementtiä poistettaessa ja yksi kokonaisuutta takaisin palauttaessa. [25] [55]

Uusia elementtejä DOM-puuhun lisättäessä ne kannattaa lisätä mahdollisimman täydellisinä. Sen sijaan että ensin asetettaisi elementti DOM-puuhun ja sen jälkeen asetettaisi sille sopivat ominaisuudet, on tehokkaampaa luoda elementti sellaisessa

muodossa että sillä on jo valmiiksi kaikki tarvittavat ominaisuudet. [25] [55]

DOM-puun elementtejä muokatessa on tehokkaampaa muokata näkymättömiä elementtejä, joille on asetettu tyyli `display: none`. Näkymättömiä elementtejä ei piirretä selaimessa, ja siten myöskään niihin tehdyt muutokset eivät aiheuta uudelleenpiirtoja. Elementtien piilottamista kannattaa hyödyntää erityisesti silloin kun muutoksia ei muulla tavoin ole mahdollista tehdä yhdellä piirrolla. Sekä elementin piilottamisesta että uudelleen esittämisestä kummastakin aiheutuu sivun piirto selaimessa. [55]

Muistin hallinta

Rikas internetsovellus voi koostua useiden sivujen sijaan vain yhdestä sivulatauksesta, jonka jälkeen sovellus saattaa kommunikoida palvelimen kanssa ainoastaan asynkronisesti AJAX-kutsuilla tai PUSH-tyyppisesti COMET-viestinnällä. Tässä tapauksessa koko sovelluksen käytön ajan selaimessa suoritetaan samaa JavaScript-sovellusta, jonka tilaa ei tyhjennetä sivunvaihdoilla.

Muistin hallinta nousee erityisen merkittäväksi yhden sivun sovelluksissa. Sovelluksen kehittäjän tulee huolehtia sovelluksen tilasta toiseen siirryttäessä, että vanhentunut sisältö poistetaan käytöstä ja siten annetaan automaattisen roskien keräyksen siivota muistinkäyttöä. Esimerkiksi huolimaton tapahtumien hallinta saattaa jättää tarpeettomia tapahtumien seuraajia koko sovelluksen elinkaaren ajan. Siten sovellus muuttuu käytettäessä jatkuvasti raskaammaksi. [54]

Kaikille JavaScript-objekteille ja DOM-elementeille tarvitaan poistokäsittely. Objektit hävitetään poistamalla kaikki viittaukset, eli asettamalla objektiin viittaavien muuttujien arvoksi null. DOM-elementtiä poistaessa pitää huomioida että kaikki siihen liittyvät attribuutit, tapahtumien käsittelijät ja sisäiset elementit poistetaan. DOM-elementtien oikeaoppisessa poistossa auttavat lukuisten eri kirjastojen poistotoiminnallisuudet, kuten jQuery.remove(). [54]

5.4 Rikkaan internetsovelluksen vaikutus verkkoliikenteeseen

Perinteisessä web-sovelluksessa toistetaan tarpeettomasti saman datan siirtoa, kun pienen muutoksen jälkeen sivupyynnöllä pyydetään palvelimelta koko sivu uudelleen. Tällöin suurin osa sivun sisällöstä, kuten navigaatioon ja sivun asetteluun liittyvät rakenteet, ovat täysin samat kuin edellisellä sivulla. Kun perinteisestä websovelluksesta siirrytään lataamaan vain muuttunut sisältö asynkronisesti, voi ensimmäisen sivun latauksen jälkeinen tiedonsiirto olla vain kymmenesosa aiemmasta. [56]

Koska rikkaan internetsovelluksen käyttöliittymä tyypillisesti virkistää tilatietoa palvelimelle, tapahtuu verkossa huomattavasti enemmän palvelinpyyntöjä. TCP-liikenteessä tämä näkyy suurempana yhteyksien määränä. HTTP-liikenne muuttuu vastaavasti harvoin tapahtuvista suurista sivupyynnöistä useiksi pieniksi pyynnöiksi.

HTTP-pyyntöjä voi tulla todella tiuhaankin, esimerkiksi sanoja ja lauseita täydentävissä tekstikentissä pyyntö lähetetään jokaisesta käyttäjän kirjoittamasta näppäinpainalluksesta. Käyttäjän syötettä ennustaa esimerkiksi Googlen haku, joka virkistää hakusanaehdotuksia jatkuvasti kun käyttäjä kirjoittaa kirjaimia hakukenttään. Jotta käyttäjä saisi välittömältä tuntuvan vasteen sovelluksen käytössä, vaaditaan verkolta pientä viivettä ja palvelimilta todella nopeaa pyynnön käsittelyä. [9]

5.5 Yhteenveto

Tässä luvussa käsiteltiin sovelluksen suorituskyvyn eri osa-alueita, jotka voidaan jakaa kolmeksi kokonaisuudeksi: sivun ja siihen liittyvien resurssien lataamiseen, sivun esittämiseen selaimessa, sekä selaimessa suoritettavan JavaScript-sovelluslogiikan suorittamiseen. Tässä luvussa esiteltyjä optimointitekniikoita hyödynnetään käytännössä seuraavan luvun mittauksissa.

Sivun ja siihen liittyvien resurssien aiheuttaman tiedonsiirron määrää voidaan optimoida tiivistämällä sisältöjä. Tekstitiedostoja voidaan pakata gzip-tiedostoiksi ja tekstisisältöjä voidaan tiivistää poistamalla tekstistä tarpeettomia tyhjiä merkkejä ja kommentteja. Tiivistämisen hyödyt tulevat parhaiten esiin JavaScript-sovelluskoodin kohdalla, kun lähdekoodin muuttujien nimet ja koodin syntaksi voidaan kääntää koneellisesti mahdollisimman tiiviiseen muotoon sovelluslogiikan muuttumatta. Sivuun liittyvien HTTP-pyyntöjen määrää voidaan rajoittaa yhdistämällä kuvatiedostoja sprite-kuviksi ja yhdistämällä JavaScript- ja CSS-tiedostoja suuremmiksi kokonaisuuksiksi.

Sivun muodostusta selaimessa voidaan nopeuttaa liittämällä sivuun vain sellaisia resursseja, joita todella käytetään sivun yhteydessä. Sivun muodostumista voidaan nopeuttaa sijoittamalla tyyli- ja JavaScript-tiedostot sopivaan kohtaan sivulla, tyylit sivun alkuun ja JavaScript sivun loppuun. Näin voidaan mahdollistaa osittaisen sivun esittäminen jo sisällön latautuessa, sekä vältytään tarpeettomilta sivun uudelleenpiirroilta.

JavaScript-sovelluskoodin suorituskykyyn vaikuttaa käytetyn selaimen ja laitteiston tehokkuuden lisäksi tehokkaaksi todistettujen ohjelmointitapojen noudattaminen ja selainrajapintojen tehokas hyödyntäminen. Selainrajapintojen harkitulla käytöllä voidaan vaikuttaa esimerkiksi selaimen suorittamien sivun piirtojen määrään.

Luku 6

Mittaukset

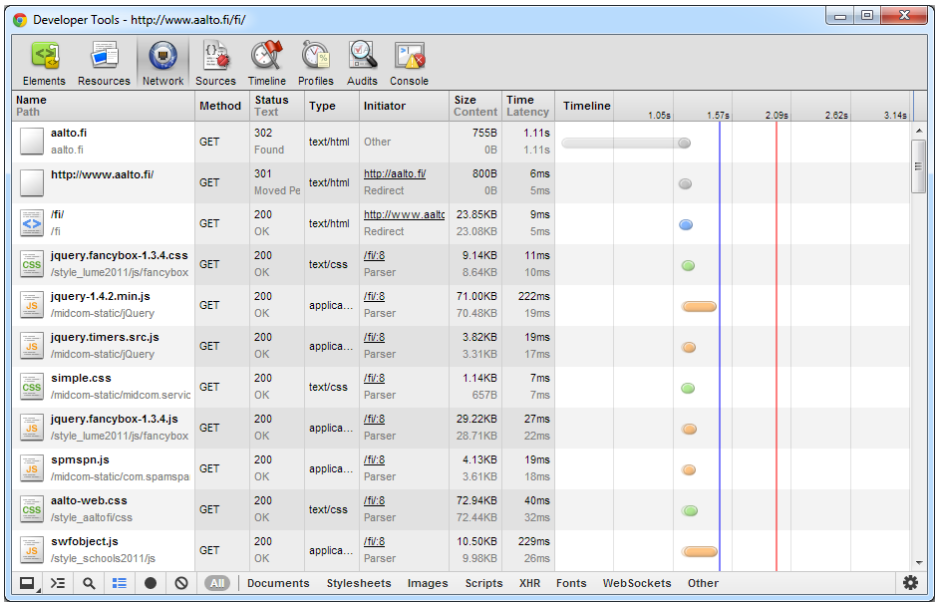
Tässä luvussa esitellään työkalut suorituskyvyn mittaamiseen ja mitataan suorituskykyä kahdessa erillisessä tapauksessa. Luvussa 6.2 optimoidaan julkisen Aalto.fi-sivuston resurssien käyttöä ja tehdään mittauksia eri optimointimenetelmien yhteydessä. Luvussa 6.3 verrataan web-sovelluksen synkronisen ja asynkronisen version välisiä eroja suljetussa ympäristössä.

6.1 Työkalut suorituskyvyn mittaamiseen ja parantamiseen

Selainten kehittäjätyökalut auttavat monissa web-sovelluksen suorituskykyyn liittyvissä selvityksissä. Suosituimpien selainten uusimmissa versioissa on mukana kehittäjien työkaluja (eng. Developer Tools) sisäänrakennettuna. Lisäksi selaimiin on saatavilla laajennuksina suorituskykyselvityksissä auttavia liitännäisiä kuten Firebug, Yahoo YSlow ja Google PageSpeed Insights. Suorituskykyselvityksiä on mahdollista tehdä ilman asennuksia erilaisten verkkopalveluiden avulla, jolloin voidaan samalla hyödyntää sivupyyntöjä useista maantieteellisistä sijainneista. Yksi sivuston suorituskykyä analysoiva ja parannusehdotuksia tarjoava palvelu on Pingdom Tools.

Verkkoliikenteen seurantaa on tyypillisesti tehty pakettitasolla Wiresharkin kaltaisilla työkaluilla, mutta web-kehittäjän selvitystyössä vastaavan tiedon saa huomattavasti selkeämmin ja tehokkaammin kehittäjätyökalujen Verkko-osiosta. Esimerkiksi Google Chromen kehittäjätyökalujen Verkko-välilehdeltä on nähtävissä mitä tiedostoja sivun lataukseen liittyy, missä järjestyksessä tiedostot ladataan, kauanko lataus kestää ja mitä tiedostoja ladataan rinnakkain. Lisäksi tiedostokohtaisesti voidaan tarkastella ajanottoa, palvelinpyynnön ja vastauksen HTTP-otsikkotietoja, käytettyjä evästeitä sekä voidaan tarkastella vastauksena saatua sisältöä. Otsikkotietojen avulla voidaan selvittää esimerkiksi oliko sisältö gzip-pakattu tai tuliko se välimuistista. Esimerkki Google Chromen kehittäjätyökalujen verkko-osiosta sivun aalto.fi sisällöllä esitetään kuvassa 6.1.

Kehittäjätyökaluilla on mahdollista tehdä myös verkon käytön ja sivun suorituskyvyn auditointeja, joiden avulla saadaan selkeitä ehdotuksia tärkeimmistä kehi-

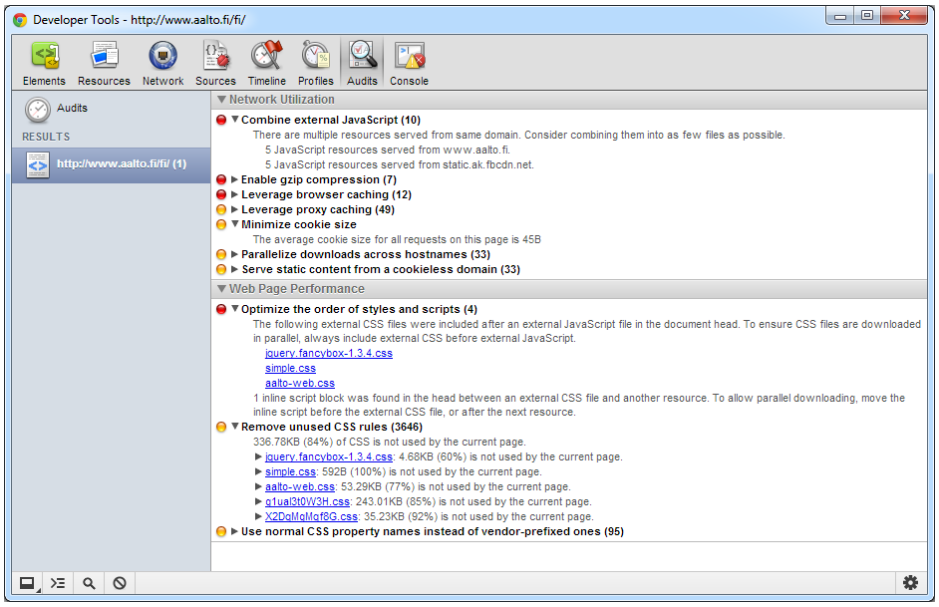


Kuva 6.1: Google Chrome kehittäjätyökalut: Verkko-osio Aalto.fi -sivustolla.

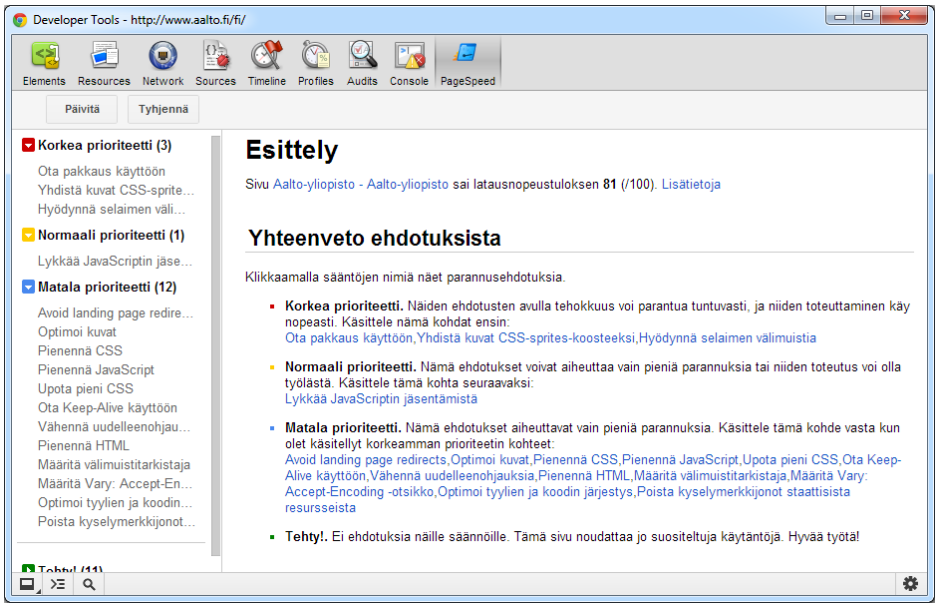
tyskohteista. Auditointeja voi tehdä paitsi sisäänrakennetuilla kehittäjätyökaluilla, myös erillisillä laajennuksilla kuten YSlow ja PageSpeed Insights. Auditoinnit esittävät sivukohtaisesti miten sivuston suorituskykyä voitaisi parantaa, sekä verkkoliikenteen että sivunmuodostuksen osalta. Kuvat 6.2 ja 6.3 esittävät Google Chromen kehittäjätyökalun ja PageSpeed Insights -laajennuksen ehdotuksia miten aalto.fi -sivun latautumisnopeutta voitaisi parantaa.

Kehittäjätyökalujen parannusehdotuksien toteuttamiseen on vastaavasti useita työkaluja. JavaScript- ja CSS-tiedostojen tiivistämiseen on tarjolla lukuisten muiden ohella Yahoo YUI Compressor, Dojo Shrinksafe sekä Douglas Crockfordin JSMin. Kuvien pakkauksen optimointia tekee Yahoon Smush.it, mikäli ei halua tehdä optimointia itse kuvankäsittelyohjelmalla. Kuvista voi tehdä sprite-koosteita lukuisilla web-palveluilla, mutta koska näiden palveluiden kyky hahmottaa kuvasisältöjä on rajallinen, päästään parhaaseen lopputulokseen usein tekemällä sprite-kuvat itse kuvankäsittelyohjelmalla.

Verkkoliikenteen lisäksi sovelluksen suoritukseen vaikuttaa merkittävästi JavaScript-koodin ja CSS-tyylimäärittelyiden suorittaminen sekä sivun piirtäminen selaimessa. Kehittäjätyökalujen profiloinnilla voidaan kerätä tilastotietoa haluttuna ajankohtana, esimerkiksi sivua ladattaessa tai tiettyä toimintoa suoritettaessa. JavaScript suoritinprofiloinnilla saadaan selville kuinka paljon eri funktiot vaativat aikaa suorittimella, ja voidaan siten selvittää sovelluksen hitaimpia toimintoja. CSS-profiloinnin tuloksena saadaan eri CSS-valitsimiin käytetty aika ja sivun elementtien osumien määrä. Siten voidaan selvittää mitkä valitsimet ovat raskaimpia ja mitä ei käytetä sivulla lainkaan. Kehittäjätyökalujen aikajana (eng. Timeline) antaa graafisen esityksen kaikista sivun tapahtumista aikajanalla. Aikajanan avulla voidaan tarkkailla tiedostojen latauksen, JavaScript-koodin suorittamisen sekä sivun muo-



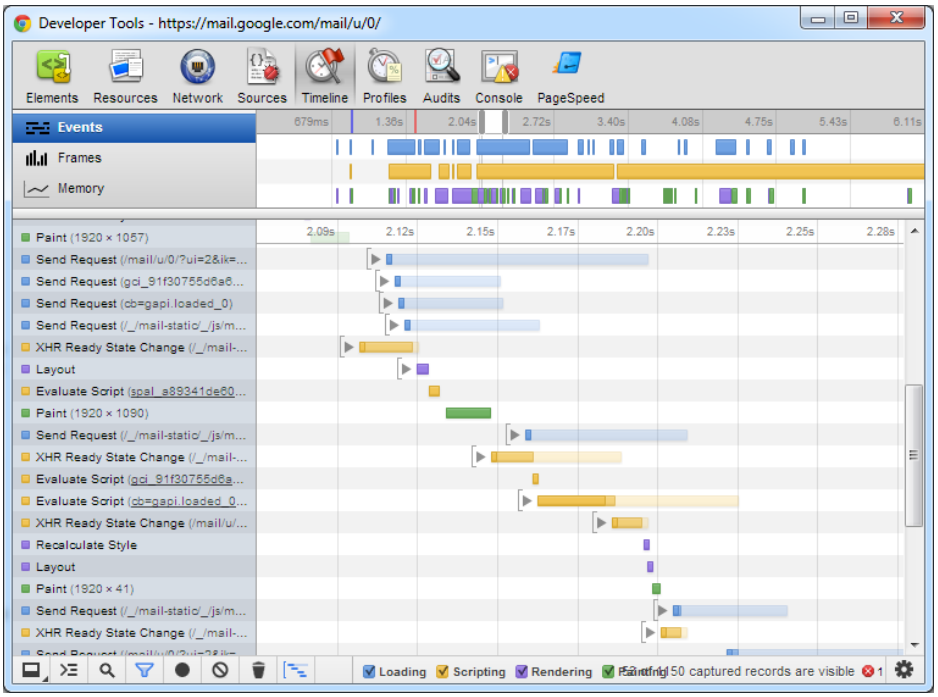
Kuva 6.2: Google Chrome -selaimen kehittäjätyökalut: Audit-osion ehdotukset Aalto.fi -sivustolla 6.11.2012.



Kuva 6.3: Google PageSpeed Insights -auditointi Aalto.fi -sivustolla 6.11.2012.

dostamisen vaatimaa aikaa ja muistinkäyttöä sivun eri vaiheissa. Sen avulla voidaan myös selvittää milloin selain piirtää sivun uudelleen esimerkiksi sivuun tehtyjen muutosten seurauksena. Kuvassa 6.4 esitetään aikajana tietyllä ajanhetkellä Gmail-palvelua avatessa.

Tässä työssä keskitytään web-sovelluksen käyttöön tietokoneella, mutta myös



Kuva 6.4: Google Chrome -selaimen kehittäjätyökalut: Aikajana Gmail-palvelun käytöstä 7.11.2012.

osassa mobiililaitteista kehittäjätyökalujen käyttö on mahdollista. Mikäli tarvitaan tietoa web-sovelluksen käytöstä puhelimella, selaimet kuten Google Chrome, Opera ja iOS-laitteiden Safari tarjoavat mahdollisuuden kehittäjätyökalujen etäkäyttöön. Puhelin liitetään tietokoneeseen USB-kaapelilla, jonka jälkeen kehittäjätyökaluja voidaan käyttää tietokoneella etänä samalla kun puhelinta käytetään web-sovelluksen käyttöön. Mikäli käytössä oleva selain ei tue kehittäjätyökalujen etäkäyttöä, on saatavilla JavaScript-kirjastoja joiden avulla voidaan lähettää tietoja mobiiliselaimesta palvelimelle tutkittavaksi.

6.2 Resurssien optimointi

Esimerkkitapauksena suorituskykyparannuksille käytettiin Aalto-yliopiston suomenkielistä etusivua osoitteessa `http://www.aalto.fi/fi/`. Sivusta käytettiin versiota joka oli julkisesti toiminnassa 15.11.2012. Sivusta ja siihen liittyvistä resursseista otettiin kopio, josta poistettiin viittaukset ulkopuolisiin Googlen ja Facebookin palveluihin. Kopiosivun palvelinympäristönä käytettiin Espoossa konesalissa sijaitsevaa Linux-palvelinta Apache-palvelinohjelmistolla. Asiakasyhteydet otettiin Vantaalta yksityisasiakkaille suunnatulla kaapelimodeemiyhteydellä, jonka nopeudeksi ilmoitetaan 40 Mbit/s. Asiakaslaitteistona käytettiin tavanomaista muutaman vuoden ikäistä Windows 7 -työpöytätietokonetta, joka oli varustettu 4 Gt keskusmuistilla ja Intel Core 2 Duo -kaksiydinprosessorilla. Selaimena käytettiin Google Chrome

-selaimen versiota 23.

Aalto-yliopiston etusivulla JavaScript-koodia on sijoitettu sekä sivun head-osioon ennen tyylejä, että eri kohtiin sivun sisällössä. Alkuperäisellä sivulla head-osion tyylitiedostot ja JavaScript-tiedostot ladataan palvelimelta tehokkaasti rinnakkain, mutta sivun sisällön kuvien lataus viivästyy, koska heti sivun alussa käsitellään JavaScript-koodia. Kuvat ladataan koodin suorituksen jälkeen rinnakkain, jonka jälkeen DOM-sisällön lataus on valmis. Tämän liipaisemana suoritetaan JavaScript-koodia, joka on määrätty suoritettavaksi kun dokumentti on valmis. Sivun lataus on kokonaisuudessaan valmis keskimäärin ajassa 740 ms. Sivun lataamiseen liittyi 42 HTTP-pyyntöä ja 451,81 kt tiedonsiirtoa. DOM oli valmis keskimäärin ajassa 432 ms. Lisäksi sivuston navigaatiota käytettäessä ladataan 6 kpl taustakuvia, ja siten sivun käytön aikana syntyy yhteensä 48 HTTP-pyyntöä.

Aluksi kaikki skriptit siirrettiin aivan sivun loppuun ja vertailtiin sivun resurssien latausta alkuperäisen ja muutetun sivun välillä. Koska kyseessä on hyvin yksinkertainen sivu, ei kokonaisajassa ollut suurta eroa. Käytännössä käyttäjä kokee kummankin sivun latautuvan välittömästi. Muokatulla sivulla saatiin kokonaislatausajaksi keskimäärin 556ms ja DOM oli valmis keskimäärin ajassa 378 ms. Sovelluksen JavaScript-koodiin lisättiin hidas laskentatoimenpide, jonka avulla voitiin havaita että alkuperäinen sivu näkyy tyhjänä laskentatoimenpiteen loppuun asti, kunnes kokonaan valmis sivu ilmestyy näkyviin. Kun koodi siirrettiin sivun loppuun, ilmestyi sivun tyylitelty runko välittömästi, ja ainoastaan sisällön kuvat piirtyivät sivulle vasta laskentatoimenpiteen jälkeen. Kaikki sivun tekstit olivat käyttäjän luettavissa koko laskutoimenpiteen ajan, ja myös kuvatiedostot ladattiin palvelimelta ennen skriptin suoritusta.

Aalto.fi-sivuston etusivua ja siihen liittyviä tekstimuotoisia resursseja ei pakata palvelimella. Kopiosivusto pakattiin Apachen `mod_deflate` -laajennuksella ja näin etusivun html-tiedosto kutistui 23,53 kilotavusta 6,42 kilotavun kokoiseksi, pienentäen tiedostokokoa 73 %. Sivuston tyylit sisältävän `aalto-web.css` -tiedoston muutos oli 72,67 kilotavusta 11,59 kilotavuun (84 % pienempi) ja minimoitu `jquery-1.4.2.min.js` JavaScript-tiedosto kutistui 70,73 kilotavusta 24,32 kilotavuun (66 % pienempi). Kun kaikki sivuun liittyvät tekstitiedostot gzip-pakattiin palvelimella, putosi sivun lataukseen liittyvä tiedonsiirto 286,61 kilotavuun, joka on 163,19 kilotavua (57 % pienempi) vähemmän kuin alkuperäinen sivukokonaisuus. Sivu latautui keskimäärin ajassa 494 ms ja DOM oli valmis ajassa 385 ms. Apache-palvelimen gzip-pakkaus asetettiin päälle lisäämällä seuraava määritys palvelimen konfiguraatioon:

```
AddOutputFilterByType DEFLATE text/html text/css text/javascript
```

Staattiset tiedostot on mahdollista pakata ennen palvelimelle siirtoa, mikäli palvelinohjelmisto ei tue pakkausta tai jos halutaan keventää palvelimen kuormitusta. Etusivun HTML-tiedosto pakattiin 7-zip -ohjelmalla gzip-muodossa ja deflate-algoritmilla, käyttäen sekä normaalia pakkausta että mahdollisimman tehokasta Ultra-pakkausta. Tiedostokoot olivat vastaavasti 6,32 kt ja 6,25 kt, eli näin saavutettiin hieman korkeampi pakkaustaso kuin Apachen `mod_deflate` laajennuksella.

Tiedostot todettiin Google Chrome -selaimessa toimivaksi, vastaavasti kuin Apachen pakkaamat tiedostot. Mikäli tiedostot pakataan itse, tulee huomioida että pakatut tiedostot nimetään kuten alkuperäiset tiedostot, ilman `.gz` -päätettä. Lisäksi palvelimen pitää lähettää HTTP-otsikko "`Content-encoding: gzip`", jotta selain ymmärtää sisällön gzip-pakatuksi.

Seuraavaksi sivun pienistä ja yleiskäyttöisistä kuvatiedostoista luotiin spritekuva. Yhdistämällä sivun harmaat ikonit yhdeksi kuvaksi, saadaan HTTP-pyyntöjen määrää vähennettyä 8 kpl. Kun navigaation taustakuvat korvataan Sprite-kuvalla, saadaan HTTP-pyyntöjen määrää vähennettyä 6 kpl. Jos taustakuvat korvataan CSS-liukuväreillä, pystytään välttämään kaikki 7 HTTP-pyyntöä. CSS-liukuvärejä käyttäessä tulee huomioida, että vanhat selaimet eivät osaa esittää niitä. Vanhojen selaimien tukemiseksi voidaan määritellä tausta ensin kuvana ja vasta sen jälkeen korkeammalla prioriteetilla CSS-liukuvärinä. Vanhat selaimet ohittavat tyylisäännöt joita ne eivät osaa esittää, ja näyttävät taustat kuvina. Näin kaikki selaimet saadaan tuettua ja uusimmat selaimet näyttävät sivun mahdollisimman tehokkaasti. Kuvamuutoksilla ja nykyaikaisella selaimella HTTP-pyyntöjen määrä saatiin laskettua 33 pyyntöön.

Sivun suurin JavaScript-tiedosto, jQuery-kirjasto, on valmiiksi minimoidussa muodossa kooltaan 70,48 kt. Täysi versio jQuery-kirjastosta olisi kooltaan 160 kt, joten säästö on merkittävä 89,52 kt (56 %). Muut sivuun liitetyt JavaScript-tiedostot minimoimalla säästetään yhteensä 10,3 kt. CSS- ja HTML-tiedostojen minimointi ei ole yhtä tehokasta, koska niiden sisältöön ei voida vaikuttaa muuten kuin tyhjien merkkien ja kommenttien poistolla. Sivun tyylitiedoston minimoimalla voidaan säästää 13,8 kt (17 %). Sivun ainoan HTML-tiedoston minimoinnilla voidaan säästää 3,6 kt (15 %).

Kaikki edellä esitetyt mittaukset on tehty ilman selaimen välimuistia. Resurssien tallennus selaimen välimuistiin auttaa poistamaan tarpeettomia HTTP-pyyntöjä, jos käyttäjä vierailee sivulla useasti tai avaa sivustolta useita samoja resursseja käyttäviä sivuja. Ensimmäistä kertaa sivulla vieraileva ei hyödy välimuistista, mutta toisen kerran sivua ladattaessa hyöty on huomattavan suuri. Selaimen välimuistia voidaan ohjata palvelimelta HTTP-otsikolla `Cache-Control`. Sen avulla voidaan antaa resursseille korkein sallittu välimuistitusaika, jonka jälkeen selain lataa vanhentuneet resurssit uudelleen palvelimelta. Kaikille kuville sekä CSS- ja JavaScript-tiedostoille asetettiin maksimi-ikä 10 vuorokautta. Apache-palvelimella max-age voidaan määrätä lisäämällä listauksen 6.1 mukainen määritys palvelimen konfiguraatioon. Määritys asettaa maksimi-iän niille tiedostoille, joiden tiedostopääte sisältyyn annettuun listaan.

Listaus 6.1: `Cache-control`-otsikkotiedon lisäys Apache-palvelimen konfiguraatioon.

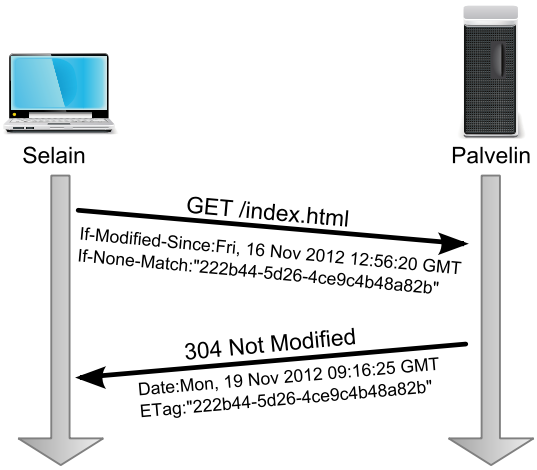
```
1 <filesMatch "\.(jpg|jpeg|png|gif|js|css|swf)$">
2   Header set Cache-Control "max-age=864000, public"
3 </filesMatch>
```

Kun dynaaminen etusivu ja sen staattiset resurssit ovat saatavilla selaimen väli-

Taulukko 6.1: Eri optimointimenetelmien vaikutukset suorituskykyyn Aalto-yliopiston etusivulla. Ajat ovat viiden sivulatauksen keskiarvoja.

	HTTP (kpl)	Tiedonsiirto (kt)	Resurssit (ms)	DOM (ms)	Sivu valmis (ms)
Alkuperäinen sivu	42	451,81	733	432	740
HTML sijoittelu	42	451,81	554	378	556
Pakkaus (gzip)	42	286,61	492	385	494
Sprite-kuvat	33	450,06	666	466	688
Minimointi (JS+CSS)	42	424,11	669	390	675
Kaikki yhdessä	33	281,16	445	392	456
Selaimen välimuisti	1	0,16	124	212	213

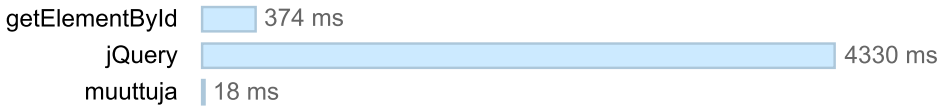
muistista, tarvitaan palvelimelle vain yksi HTTP-pyyntö, jolla haetaan dynaamista sisältöä sisältävä HTML-tiedosto. Kuvan 6.5 mukaisesti käyttäjän selain ilmoittaa kuinka vanha versio sivusta sillä on käytettävissä, ja jos sivu on muuttumaton, voi palvelin palauttaa ainoastaan HTTP-otsikon `304 Not Modified`. Koko sivun lataukseen tarvitaan vain yksi HTTP-pyyntö ja 159 tavun tiedonsiirto. Sivun resurssit saadaan ladattua keskimäärin ajassa 124 ms ja DOM on valmis vasta resurssien latauksen jälkeen ajassa 212 ms. Sivun muodostus on valmis ajassa 213 ms, joka on 71 % nopeampi kuin alkuperäisellä sivulla. Mikäli sivu sisältäisi dynaamista sisältöä, jonka välimuistiin tallentaminen estetään, ei välimuistilla saavutettaisi yhtä suurta hyötyä. Välimuistin vaikutusta suurilla dynaamisilla sisällöillä mitataan luvussa 6.3.



Kuva 6.5: Selaimen välimuistiin tallennetun muuttumattoman tiedoston kysely palvelimelta.

Eri menetelmät ja niitä vastaava suorituskyky on listattu taulukossa 6.1. Skriptien ja CSS-tiedostojen sijoittelu HTML-tiedostossa oikein vaikuttaa käyttökokemukseen erityisesti raskaammilla sivuilla, mutta ei näy lukuina suurena parannuksena kun kyseessä on kevyt sivu. Gzip-pakkaus auttaa merkittävästi tiedonsiirron vähentämisessä ja mikäli käytössä oleva www-palvelin tukee pakkausta, on sen käyttöönotto vaivatonta. Sprite-kuvat ovat kehittäjälle työläämpi muutos, koska se edellyttää kuvatiedostojen ja tyylisääntöjen uudistamista. Erityisesti graafisissa sovelluksissa joissa on käytössä paljon pieniä graafisia elementtejä, kuten ikoneja ja painikkeita, voivat sprite-kuvat auttaa vähentämään HTTP-pyyntöjen määrää merkittävästi. CSS-tyylitiedostojen minimointi tuo hyvin rajallisen hyödyn, koska valitsimia ja määrittelyitä ei voida nimetä uudelleen. Sen sijaan JavaScript-tiedostoissa hyöty on merkittävä, kun koodin muuttujat voidaan nimetä mahdollisimman tiiviisti. Tässä tapauksessa JavaScript-minimointi ei tuonut suurta suorituskykyparannusta, koska käytössä oli valmiiksi minimoitu versio jQuery-kirjastosta ja sivun minimoitavissa oleva JavaScript-koodi oli hyvin yksinkertaista. Kun kaikki menetelmät otetaan yhdessä käyttöön, saavutetaan selvä parannus kaikissa mitatuissa arvoissa jopa tämän tapauksen yksinkertaisella sivulla. Hyöty on suurempi, kun samat optimoinnit toteutetaan sivulla, jolla on enemmän grafiikkaa ja esityslogiikkaa.

Selaimen DOM-kyselyiden raskautta testattiin pyytämällä DOM-rajapinnalta yksi HTML-sivun elementti id-tunnisteella `main-right`. Kyselylle tehtiin kolme eri tapausta, joista ensimmäisessä elementti haetaan toistuvasti DOM-metodilla muuttujaan `getElementById('main-right')`, toisessa vastaavasti käyttäen `jQuery('#main-right')` -hakua. Viimeisessä testissä haetaan DOM-elementti muuttujaan vain kerran ja sijoitetaan se toistuvasti muuttujasta toiseen muuttujaan. Toimintoja suoritettiin viiden miljoonan kerran toistona. Testi toistettiin kymmenen kertaa ja tuloksista laskettiin keskiarvo. Aikatiedot saatiin Google Chromen `console.time()` -toiminnolla. Kuva 6.6 esittää mittauksen tulokset. Tuloksista voidaan selvästi havaita että toistuvissa DOM-toimenpiteissä on hyödyllistä käyttää peruskirjaston sijaan suoraa DOM-kutsua ja toistuvan DOM-rajapinnan käytön sijaan on järkevää tallentaa DOM-kyselyn tulos paikalliseen muuttujaan.



Kuva 6.6: Eri tavat hakea DOM-puun tieto (5 000 000 toistoa).

Tämän luvun mittauksien perusteella suurin vaikutus sivun latausnopeuteen oli käyttäjän selaimen välimuistilla, jonka avulla alkuperäinen sivu latautui alle puolessa siitä ajasta, joka saavutettiin kaikki muut optimointikeinot yhdistämällä. Selaimen välimuisti ei kuitenkaan vaikuta latausnopeuteen ensimmäistä kertaa sivulla vieraillessa. Sivun sisällön optimointikeinoista latausnopeutta tehokkaimmin paransi gzip-pakkaus. Pakkaus oli myös selvästi tehokkain keino sivuun liittyvän tiedonsiirron vähentämiseksi. Sprite-kuvien hyödyntämisellä saavutettiin alhaisin HTTP-pyyntöjen määrä, mutta vaikutus sivun latausnopeuteen oli vähäisin. Kaikki mitatut

optimointikeinot paransivat sivun suorituskykyä ja parhaaseen tulokseen päästiin käyttämällä kaikkia optimointikeinoja yhdessä. Mittauksien perusteella sovelluksen suorituskyvyn kannalta on tärkeää käyttää DOM-rajapintaa harkiten, ja toistuvien DOM-pyyntöjen sijaan tieto kannattaa ehdottomasti tallentaa paikalliseen muuttujaan. Lisäksi raskaissa operaatioissa kannattaa harkita suoraa selainrajapintojen käyttöä peruskirjaston tarjoamien toimintojen sijaan.

6.3 Asynkronisen ja synkronisen välinen ero

Tähän työhön liittyvässä projektissa uudistetaan web-sovellus hyödyntämään aiempaa laajemmin asynkronista tiedonsiirtoa, selaimessa suoritettavaa käyttöliittymälogiikkaa ja REST-rajapintaa. Työn yhteydessä oli mahdollista tutkia vanhan ja uuden sovelluksen eroja uuden sovelluksen varhaisessa kehitysvaiheessa.

Uusi sovellus hyödyntää palvelinpuolella Groovy-ohjelmointikieltä ja Grails-sovelluskehitysympäristöä, kun vanha sovellus perustuu JavaEE- ja JSP-tekniikoihin. Sovellusympäristöjen erojen vuoksi uuden sovelluksen taustalla tapahtui käyttöliittymätekniikan lisäksi paljon muitakin suorituskykyyn vaikuttavia muutoksia. Uudessa ja vanhassa sovelluksessa käytettiin samaa tietokantaa ja identtistä palvelinsovelluksen EJB-tason toteutusta. Kumpaakin sovellusta ajettiin Oracle WebLogic -palvelimella samalla fyysisellä palvelinlaitteistolla. Palvelinlaitteistona toimi Windows 7 -työpöytätietokone 8 Gt keskusmuistilla, Intel Core i5 -suorittimella ja SSD-massamuistilla. Ympäristön asettamien rajoitteiden vuoksi selainta käytettiin samalla laitteella kuin palvelinsovellusta, joten sivuja ja niiden resursseja palvelimelta ladattaessa ei käytetty lähiverkkoa eikä internet-yhteyttä. Selaimena käytettiin Google Chromen versiota 23. Uuden sovelluksen varhaisen kehitysvaiheen vuoksi sitä ei ollut mahdollista ajaa todellista käyttötapausta vastaavassa ympäristössä internet-yhteyttä käyttäen. Tästä syystä verkkoliikenne on testeissä nopeampaa kuin todellisessa käyttötapauksessa.

Ympäristöjen ja sivun rakenteiden välisiä eroja pyrittiin selvittämään mittaamalla sisällöltään toisiaan vastaavan aloitussivun avaamisnopeutta uudessa ja vanhassa

Taulukko 6.2: Aloitussivun latausnopeus vanhalla ja uudella sovelluksella. Ajat ovat viiden latauskerran keskiarvoja.

	Palvelin-käsittely (ms)	HTML-tiedonsiirto (ms)	HTTP-pyynnöt (kpl)	Tiedonsiirto yht. (kt)	Sivu valmis (ms)
Vanha sovellus	71	9	73	505	434
Uusi sovellus	52	1	23	273	274
Vanha + välimuisti	71	9	1	4,19	257
Uusi + välimuisti	52	1	1	2,34	209

sovelluksessa. Tulokset esitetään taulukossa 6.2. Aloitussivu sisältää yksinkertaisen hakulomakkeen, jonka täyttämällä saadaan hakukriteereitä vastaava hakutuloslistaus. Kun vanhaa sovellusta käytettäessä selaimesta on estetty välimuisti, sivu latautuu keskimäärin ajassa 434 ms. Lataukseen liittyy 73 HTTP-pyyntöä ja 505 kt tiedonsiirtoa. Palvelin käsittelee sivupyyntöä 71 ms ja kun HTML-sivu on muodostettu palvelimella, sen tiedonsiirto vie 9 ms ajan. HTML-sivu on gzip-pakattu ja sen koko pakattuna on 4,19 kt (pakkaamaton 21,17 kt). Kun aloitussivua käytetään selaimen välimuistia hyödyntäen, palvelimelta ladataan vain HTML-sivu ja uusi sivu muodostetaan selaimessa ajassa 257 ms.

Uudella aloitussivulla lataukseen liittyy 23 HTTP-pyyntöä ja 273 kt tiedonsiirtoa. Sivun lataus kestää kokonaisuudessaan 274 ms. Palvelin käsittelee HTML-sivun pyyntöä keskimäärin 52 ms ja tiedonsiirto tapahtuu alle 1 ms ajassa. Pakatun HTML-sivun koko on 2,34 kt (pakkaamaton 7,8 kt). Selaimen välimuistia hyödyntäen sivu latautuu ajassa 209 ms. Koska vanhan sovelluksen sivu on HTML-merkintätavan ja kuvien määrän osalta raskaampi, on huonompi suorituskyky odotettavissa silloin kun kaikki resurssit ladataan. Selaimen välimuistia hyödynnettäessä uuden ja vanhan sovelluksen suorituskyky on hyvin samankaltainen ja ympäristöt ovat siten vertailukelpoisia.

Vanhassa ympäristössä lomake lähetetään palvelimelle HTTP GET -sivupyyntönä ja palvelimelta saadaan vastauksena lomakkeen ja hakutulokset sisältävä valmis tulossivu. Palvelin muodostaa HTML-tulossivua 1,77 s ajan ja sen tiedonsiirto kestää 0,54 s. Valmis HTML-tulossivu sisältää 637 hakutulosta ja se on kooltaan gzip-pakattuna 34,74 kt. Sivun pakkaamaton sisältö on kooltaan huomattavasti suurempi, 928,05 kt. Tulossivun muodostuminen mitattiin kahdessa tapauksessa. Kun käyttäjän selaimessa ei ole välimuisti käytettävissä, tulossivu on valmis 3,67 sekunnissa. Normaalisti käyttäjillä on välimuisti käytettävissä ja sivun resurssit valmiiksi välimuistissa hakutulossivua ladattaessa. Välimuistia hyödynnettäessä latausaika on 2,71 sekuntia ja palvelimelta ladattavaa sisältöä on vain HTML-sivun 34,74 kt kokoinen sisältö.

Uudessa ympäristössä lomakkeen lähetys toteutetaan AJAX-kutsuna REST-rajapinnalle. Rajapinta palauttaa tulokset tiiviissä JSON-muodossa ja aloitussivun selainsovellus muodostaa tuloksista HTML-esityksen, joka lisätään sivun pohjalle vastaavaan kohtaan kuin vanhan sovelluksen vastaussivussa. Käyttäjä pysyy koko ajan samalla sivulla, jonka sisältö muuttuu hakutoimintoja käytettäessä. Vastaavat 637 hakutulosta ovat JSON-muodossa gzip-pakattuna kooltaan 25,98 kt ja pakkaamattomana 245 Kt. Valmiiksi muotoiltuun sivuun verrattaessa koko on pakattuna 25 % pienempi ja pakkaamattomana 74 % pienempi. Hakutuloksia pyydettäessä ainoa ladattava resurssi on dynaaminen JSON-tiedosto, jonka välimuistiin tallentamista ei sallita. Välimuistin käyttö ei siten vaikuta pyynnön raskauteen tässä tapauksessa, kun hakutuloslistan muodostamiseen ei liity kuvia tai muita sivun ulkopuolisia resursseja. JSON-tiedoston pyyntö ja vastauksen muodostaminen palvelimella kesti 1001 ms ja kokeen ympäristössä vastauksena lähetettävän tiedoston verkkoliikenteen osuus oli alle 1 ms. Tulosten esittämiseksi tarvittavat sivun muutokset kestivät keskimäärin 70 ms ja kokonaisaika hakutulosten näyttämiseksi AJAX-tekniikalla on siten keskimäärin 1071 ms. Hakutuloksiin liittyvien mittausten tulokset esitetään

Taulukko 6.3: Hakutulosten latausnopeus vanhalla ja uudella sovelluksella. Ajat ovat viiden latauskerran keskiarvoja ja käytössä on gzip-pakkaus.

	Palvelin-käsittely (ms)	HTML-tiedonsiirto (ms)	HTTP-pyynnöt (kpl)	Tiedonsiirto yht. (kt)	Sivu valmis (ms)
Vanha sovellus	1 768	536	76	541	3 674
Vanha+välimuisti	1 886	558	1	35	2 710
Uusi sovellus	1 001	1	1	26	1071

Taulukko 6.4: Hakutulosten latausnopeus vanhan sovelluksen tuotantoympäristössä VPN-yhteydellä. Ajat ovat viiden latauskerran keskiarvoja.

	Palvelin-käsittely (ms)	HTML-tiedonsiirto (ms)	HTTP-pyynnöt (kpl)	Tiedonsiirto yht. (kt)	Sivu valmis (ms)
Pakattu (gzip)	2 184	388	79	556	4 634
Pakattu + välimuisti	2 106	554	1	36	3 114
Pakkaamaton	2 054	2 986	79	2 590	6 902
Pakkaamaton + välimuisti	2 182	1 592	1	942	4 100

taulukossa 6.3.

Uuden ympäristön testeissä huomattiin gzip-pakkauksen tärkeys. Pakkaus on erittäin hyödyllistä paitsi staattisilla JavaScript- ja CSS-tiedostoilla, myös dynaamisesti muodostettavilla HTML- ja JSON-tiedostoilla. Pakkaamaton JSON-tuloslista oli moninkertaisesti suurempi kuin kaikki hakutulokset sisältävä ja valmiiksi HTML-muotoiltu sivu gzip-pakattuna. Jotta gzip-pakkauksen vaikutusta voitaisiin paremmin arvioida todellisessa käyttötapauksessa, vanhan sovelluksen tuotantokäytössä olevaa versiota kokeiltiin gzip-pakkauksella ja ilman. Sivu on sama hakutulossivu kuin aiemmissa vanhan ympäristön mittauksissa, sillä erolla että sivu koostui nyt 623 hakutuloksesta aiemman 637 hakutuloksen sijaan. Testi suoritettiin todellisessa toimintaympäristössä ottamalla yhteys palvelimeen kotiverkosta VPN-yhteydellä. Mittaustulokset esitetään taulukossa 6.4. Pakatun ja pakkaamattoman version erot ilman välimuistin käyttöä kuvaavat sitä eroa, joka saavutetaan pakattaessa kaikki sivuun liittyvät tekstisisällöt. Välimuistia käytettäessä saadaan selvitettyä välimuistin tuoma suorituskykyparannus kun sivu koostuu suuresta määrästä dynaamista sisältöä, jota ei voida tallentaa välimuistiin. Välimuistia käytettäessä palvelimelta ladataan vain dynaaminen HTML-sivu, ja mittauksissa voidaan siten keskittyä pelkän HTML-tiedoston pakkauksen vaikutukseen. Kaikissa tapauksissa

suuri osa kokonaisajasta kuluu sivua palvelimella muodostaessa, jonka kesto on kaikissa tapauksissa samaa suuruusluokkaa: noin 2 sekuntia. Kun kaikki resurssit ladataan palvelimelta ilman välimuistia, pakattu sivu latautuu 33 % nopeammin kuin pakkaamaton. Selaimen välimuistia hyödynnettäessä pakattu sivu on 24 % nopeampi. Pakkaamattomalla sivulla välimuistin avulla saadaan 41 % nopeampi latausaika, kun edellisen luvun mittauksissa staattisella HTML-sivulla päästiin 71 % nopeusparannukseen. Pakatulla sisällöllä selaimen välimuisti nopeuttaa sivun latausta 33 %.

Ensimmäisissä testeissä huomattiin myös JSON-tulosten sisällön optimoinnin tärkeys, kun käsitellään suurta tulosjoukkoa. Ensimmäisen rajapintatoteutuksen lähettämä tulosdata oli kooltaan yli megatavun, koska datassa toistettiin suuria määriä tietoa jota käyttöliittymäsovellus ei koskaan käyttänyt. Jos uuden sovelluksen testit olisi suoritettu tuotantoympäristössä todelliselta asiakaskoneelta internetin välityksellä, olisi verkkoliikenteen rooli merkittävämpi. Tällöin AJAX-tekniikan edut korostuisivat, kun verkkoliikennettä on vähemmän. Testit suoritettiin yhden käyttäjän ympäristössä, todellisessa tuotantojärjestelmässä käyttäjien määrä olisi huomattavasti suurempi. Tällöin palvelin ruuhkautuu ja palvelinvastauksen muodostus hidastuu. Suurillakin käyttäjämäärillä käyttäjän laitteistoon kohdistuva kuormitus on sama kuin testitilanteessa, joten myös suurilla käyttäjämäärillä palvelimen työmäärää vähentävät AJAX-tekniikat antavat suotuisan vaikutuksen suorituskykyyn. Huomion arvoista on myös se, että sivun käyttöliittymä on uudessa ympäristössä käyttäjän käytettävissä samalla kun hakutuloksia haetaan AJAX-pyyntönä.

Aloitussivun testeillä voitiin todeta, että uuden ja vanhan sovelluksen ympäristöissä sivun latausnopeudet ovat samankaltaisia kun selaimessa käytetään välimuistia. Aloitussivun lataus oli uudessa ympäristössä noin 19 % nopeampi. Hakutulossivun lataus oli vanhassa ympäristössä huomattavasti raskaampaa kuin HTML-perusrakenteiltaan samanlaisen aloitussivun lataus. Siten ympäristön ja sivurakenteen väliset erot uuden ja vanhan sovellusympäristön välillä ovat riittävän pieniä, jotta ympäristöjä voidaan pitää vertailukelpoisina. Varsinainen synkronisen ja asynkronisen lataustavan välinen vertailu tehtiin ladattaessa aloitussivun hakuehtojen mukaisia hakutuloksia. Kun kummassakin ympäristössä käytössä on selaimen välimuisti, ladattavaksi jää ainoastaan vanhan sovelluksen tapauksessa tulokset sisältävä HTML-sivu ja uuden sovelluksen kohdalla tulokset sisältävä JSON-tiedosto. Vanhan hakutulossivun lataukseen liittyy valmiin sivun muodostaminen palvelimella, sen siirtäminen selaimeen ja koko sivun piirtäminen selaimessa. Uuden sovelluksen hakutulosten lataamiseen liittyy JSON-tulostiedoston mudostaminen, sen siirtäminen selaimeen, sivun HTML-muotoilun muodostaminen selaimessa ja lopulta muutoksien piirtäminen selaimessa. Kaikki nämä vaiheet huomioitiin mittauksissa ja hakutulosten valmistuminen asynkronisesti ladattuna on 60 % nopeampaa kuin synkronisesti ladattuna.

Luku 7

Yhteenveto

Tässä työssä esiteltiin rikkaiden internetsovellusten rakentamiseen käytettävät tekniikat perustekniikoista uusimpiin HTML5-tekniikoihin. Sovelluksen ja käyttäjän välinen vuorovaikutus paranee uusimpien tekniikoiden avulla, kun käyttöliittymän vaste saadaan nopeaksi ja käyttöliittymän visuaalinen esitys voidaan tehdä joustavasti työpöytäsovelluksen kaltaiseksi. Uusimpien tekniikoiden avulla on mahdollista siirtää sovelluksen esityslogiikka palvelimelta käyttäjän selaimeen, ja siten on mahdollista hajauttaa sovelluksen prosessointia käyttäjien laitteistoille. Salainen bisneslogiikka ja käyttäjän syötteiden lopullinen varmentaminen pitää tietoturvan vuoksi suorittaa edelleen palvelimella, koska JavaScript-sovelluksen lähdekoodi, data ja tietoliikenne ovat osaavan käyttäjän nähtävissä ja käsiteltävissä.

Selaimessa suoritettavan JavaScript-sovelluksen koodin hahmottaminen ja ylläpidettävyys muuttuu haasteelliseksi, kun sovelluslogiikka kasvaa monimutkaiseksi kokonaisuudeksi. Jotta sovellusta on vaivatonta laajentaa ja ylläpitää, tarvitaan selkeää arkkitehtuuria. Suunnittelumallit auttavat hyödyntämään JavaScript-kielen hyviä puolia ja kiertämään sen rajoitteita. Laadukkaat kirjastot ja sovelluskehykset tarjoavat sovelluksen arkkitehtuurille hyvän perustan.

Suorituskykyä voidaan parantaa optimoimalla sivuun liitettäviä resursseja niin, että resurssien kokoa ja lukumäärä saadaan pienennettyä ja käytössä on vain oleelliset resurssit. Resurssien kokoa voidaan tiivistää pakkaamalla ja minimoimalla tekstisisältöjä, sekä valitsemalla kuville sopiva tiedostomuoto ja pakkaustaso. Resurssien lukumäärää ja niiden lataamiseen tarvittavien HTTP-pyyntöjen määrää voidaan vähentää yhdistämällä tekstisisältöjä ja kuvia suuremmiksi kokonaisuuksiksi. Selaimen välimuistin tehokas hyödyntäminen mahdollistaa resurssien hakemisen paikallisesta muistista palvelimen sijaan, ja siten saadaan merkittävästi vähennettyä sekä pyyntöjen määrää, että resurssien vaatimaa tiedonsiirtoa. HTML-sivun muodostamisen tehokkuudessa oleellista on missä kohdassa tyyli- ja JavaScript-resurssit liitetään sivuun.

Harkitsematon selainrajapintojen käyttö voi johtaa huonoon suorituskykyyn. Ongelmat korostuvat entisestään kun rajapintoja käytetään JavaScript-kirjastoja hyödyntäen, ymmärtämättä mitä kirjaston toiminnot käytännössä tekevät. Työssä esiteltyjä suositeltavia ohjelmointitapoja hyödyntäen voidaan välttyä vakavimmilta suorituskykyongelmilta.

Työn mittauksilla vahvistettiin optimointikeinojen tehokkuus käytännössä. Kaikki optimointikeinot osoittautuivat sivun suorituskykyä parantaviksi. Tehokkaimmiksi keinoiksi osoittautuivat selaimen välimuistin tehokas hyödyntäminen ja kaikkien tekstisisältöjen gzip-pakkaus. Välimuistilla saavutettiin staattisella aalto.fi-sivulla 71 % ja dynaamisella hakutulossivulla 41 % nopeampi latausaika. Tekstisisältöjen pakkauksella saavutettiin ilman välimuistia kummassakin testitapauksessa 33 % nopeampi latausaika. Hakutulosten pakkaamisella saavutettiin todella suuria parannuksia tiedostokokoon, kun JSON-tulokset kutistuivat 89 % ja HTML-tulokset 96 %. Hakutuloksien esittämistä mitattiin sekä synkronisella että asynkronisella lataustavalla. Hakutulokset valmistuivat asynkronisesti JSON-tiedostona ladattuna 60 % nopeammin kuin synkronisesti hakutulossivuna lataamalla, vaikka synkronisen latauksen yhteydessä hyödynnettiin sivuun liittyvien resurssien latauksessa selaimen välimuistia.

Tämän työn avulla tullaan kehittämään olemassa olevasta web-sovelluksesta uusi, aiempaa rikkaampi versio. Uuden sovelluksen myötä tullaan siirtymään sivunvaihtoon perustuvasta käyttötavasta hyödyntämään aiempaa laajemmin asynkronisia JavaScript-tekniikoita. Sivun muodostamisen vastuuta siirretään palvelimelta käyttäjän selaimelle ja palvelinsovelluksen rooli painottuu REST-rajapintaan ja salaiseen bisneslogiikkaan. Käyttäjän selaimessa toimiva käyttöliittymäsovellus tulee hyödyntämään MVC-arkkitehtuuria ja Backbone.js-kirjastoa. Sovellus tulee sisältämään kymmeniä tuhansia rivejä JavaScript-koodia ja sen avulla tehdään raskaita bisnestoimintoja. Selainsovelluksen arkkitehtuuri ja suorituskyky ovat entistä tärkeämmässä roolissa ja nyt ne on mahdollista tehdä alusta alkaen nykyaikaisia tekniikoita hyödyntäen.

Kirjallisuutta

[1] Adobe. Adobe roadmap for the flash runtimes. `http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/flashplatform/whitepapers/flash-runtimes-roadmap.pdf`, Sept. 2012. Haettu: 10.09.2012.

[2] Arnaud, L., Philippe, L., Lauren, W., Gavin, N., Jonathan, R., Mike, C., and Steve, B. Document object model (dom) level 3 core specification. *World Wide Web Consortium (W3C) Recommendation* (2004).

[3] Barth, A. Http state management mechanism. *Internet Engineering Task Force, Request for Comments (RFC) 6265* (2011).

[4] Battle, R., and Benson, E. Bridging the semantic web and web 2.0 with representational state transfer (rest). *Web Semantics: Science, Services and Agents on the World Wide Web 6*, 1 (2008), 61 – 69.

[5] Berners-Lee, T., Fielding, R., and Masinter, L. Uniform resource identifier (uri): Generic syntax. *Internet Engineering Task Force, Request for Comments (RFC) 3986* (2005).

[6] Bos, B., Celik, T., Hickson, I., and Lie, H. Cascading style sheets, level 2 revision 1 css 2.1 specification. *W3C working draft, W3C, June* (2005).

[7] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and Yergeau, F. Extensible markup language (xml) 1.0. *World Wide Web Consortium 2009*, 11/23 (2008).

[8] Celik, T. Css basic user interface module level 3 (css3 ui). *W3C working draft, W3C, January* (2012).

[9] Crescendo. How Rich Internet Applications Impact the Network. Tech. rep., 2008.

[10] Crockford, D. Json. *Internet Engineering Task Force, Request for Comments (RFC) 4627 29* (2009).

[11] Dickinson, J. *Grails 1.1 web application development.* Packt Publishing, 2009.

[12] Farrell, J., and Nezlek, G. Rich internet applications the next stage of application development. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on* (june 2007), pp. 413 –418.

[13] Fette, I., and Melnikov, A. The websocket protocol. *IETF 6455* (2011).

[14] Fielding, R. Architectural styles and the design of network-based software architectures.

[15] Fielding, R. Rest apis must be hypertext driven. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`, 2008. Verkossa julkaistu kirjoitus. Julkaistu 20.10.2012. Haettu 5.10.2012.

[16] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext transfer protocol–http/1.1, 1999.

[17] Flanagan, D. *JavaScript: The Definitive Guide*. Definitive Guides. O'Reilly Media, 2011.

[18] Freeman, A., and Freeman, A. Silverlight. In *Introducing Visual C# 2010*. Apress, 2010, pp. 1125–1157.

[19] Garlan, D. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), ACM, pp. 91–101.

[20] Garrett, J., et al. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, 2005. Verkossa julkaistu artikkeli. Haettu 7.11.2012.

[21] Gil, B., and Trezentos, P. Impacts of data interchange formats on energy consumption and performance in smartphones. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (2011), ACM, pp. 1–6.

[22] Griffin, K., and Flanagan, C. Browser based communications integration using representational state transfer. In *Novel Algorithms and Techniques in Telecommunications and Networking*, T. Sobh, K. Elleithy, and A. Mahmood, Eds. Springer Netherlands, 2010, pp. 323–328.

[23] Hammond, J. S. Ajax or Flex?: How to Select RIA Technologies. Tech. rep., Forrester, 2006.

[24] Harjono, J., Ng, G., Kong, D., and Lo, J. Building smarter web applications with html5. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research* (Riverton, NJ, USA, 2010), CASCON '10, IBM Corp., pp. 402–403.

[25] Heng, K. Speeding up javascript: Working with the dom. `https://developers.google.com/speed/articles/javascript-dom`, 2012. Haettu: 6.11.2012.

[26] Herczeg, Z., Lóki, G., Szirbucz, T., and Kiss, Á. Guidelines for javascript programs: Are they still necessary. In *Proceedings of the 11th Symposium on Programming Languages and Software Tools (SPLST'09) and 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE'09)* (2009), pp. 59–71.

[27] Hickson, I. Server-sent events. *World Wide Web Consortium (W3C) Working Draft* (2012).

[28] Hickson, I., and Hyatt, D. Html5: A vocabulary and associated apis for html and xhtml. *W3C Working Draft* (2012).

[29] Kienle, H. It's about time to take javascript (more) seriously. *Software, IEEE 27*, 3 (may-june 2010), 60 –62.

[30] Lawton, G. New ways to build rich internet applications. *Computer 41*, 8 (aug. 2008), 10 –12.

[31] Lee, S.-W., Moon, S.-M., Jung, W.-K., Oh, J.-S., and Oh, H.-S. Code size and performance optimization for mobile javascript just-in-time compiler. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture* (New York, NY, USA, 2010), INTERACT-14, ACM, pp. 6:1–6:7.

[32] Mockapetris, P. Domain names-implementation and specifications, 1987.

[33] Mozilla Developer Network. Using server-sent events. `https://developer.mozilla.org/en-US/docs/Server-sent_events/Using_server-sent_events`, 2012. Verkossa julkaistu artikkeli. Haettu 30.11.2012.

[34] Nurseitov, N., Paulson, M., Reynolds, R., and Izurieta, C. Comparison of json and xml data interchange formats: A case study. *Department of Computer Science Montana State University–Bozeman, Montana 59715* (2009).

[35] Oracle. Javafx roadmap. `http://www.oracle.com/technetwork/java/javafx/overview/roadmap-1446331.html`, Sept. 2012. Haettu: 10.09.2012.

[36] Oreilly, t. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & Strategies, No. 1, p. 17, First Quarter 2007*.

[37] Osmani, A. Developing backbone.js applications. `http://addyosmani.github.com/backbone-fundamentals/`, 2012. Verkossa julkaistu kirja. Haettu: 11.9.2012.

[38] Osmani, A. Learning javascript design patterns. `http://addyosmani.com/resources/essentialjsdesignpatterns/book/`, 2012. Verkossa julkaistu kirja, versio 1.5.2. Haettu 13.9.2012.

[39] Peng, W., and Cisna, J. Http cookies–a promising technology. *Online Information Review 24*, 2 (2000), 150–153.

[40] Pixley, T., et al. Document object model (dom) level 3 events specification. *World Wide Web Consortium (W3C) Working Draft* (2012).

[41] PremKumar, L., and Mohan, P. *Beginning JavaFX*. Apress Series. Apress, 2010.

[42] Raggett, D., Le Hors, A., Jacobs, I., et al. Html 4.01 specification. *W3C recommendation 24* (1999).

[43] Rao, S. Css3 vs jquery animations. `http://dev.opera.com/articles/view/css3-vs-jquery-animations/`, 2012. Verkossa julkaistu artikkeli. Haettu 8.11.2012.

[44] Ratanaworabhan, P., Livshits, B., Simmons, D., and Zorn, B. Jsmeter: Measuring javascript behavior in the wild. In *Usenix Conference on Web Application Development (WebApps)* (2010).

[45] Require.js. Require.js documentation. `http://requirejs.org/docs/1.0/docs/`. Verkossa julkaistu dokumentaatio. Versio 1.0.8. Haettu 21.9.2012.

[46] Resig, J. The dom is a mess. `http://ejohn.org/blog/the-dom-is-a-mess/`, 2009. Verkossa julkaistu esitys. Haettu 8.11.2012.

[47] Richards, G., Lebresne, S., Burg, B., and Vitek, J. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not. 45*, 6 (June 2010), 1–12.

[48] Ritchie, P. The security risks of ajax/web 2.0 applications. *Network Security 2007*, 3 (2007), 4 – 8.

[49] Sharp, R. Server-sent events. `http://html5doctor.com/server-sent-events/`, 2012. Verkossa julkaistu artikkeli. Haettu 30.11.2012.

[50] Souders, S. High-performance web sites. *Commun. ACM 51*, 12 (Dec. 2008), 36–41.

[51] Souders, S. *Even Faster Web Sites*. O'Reilly Series. O'Reilly, 2009.

[52] The jQuery Project. Jquery documentation. `http://docs.jquery.com`. Verkossa julkaistu dokumentaatio. Haettu 21.9.2012.

[53] Tilkov, S., and Vinoski, S. Node.js: Using javascript to build high-performance network programs. *Internet Computing, IEEE 14*, 6 (nov.-dec. 2010), 80 –83.

[54] Van, M. Javascript memory management. `http://blog.socialcast.com/javascript-memory-management/`, 2011. Haettu 27.11.2012.

[55] Wilton-Jones, M. Efficient javascript. `http://dev.opera.com/articles/view/efficient-javascript/`, 2006. Haettu 27.11.2012.

[56] YANG, J., wei LIAO, Z., and LIU, F. The impact of ajax on network performance. *The Journal of China Universities of Posts and Telecommunications 14, Supplement 1*, 0 (2007), 32 – 34.

[57] Zakas, N. *High Performance JavaScript*. O'Reilly Series. O'Reilly, 2010.

[58] Zakas, N. Managing javascript objects. *MSDN Magazine* (2010). Verkossa julkaistu artikkeli. Julkaistu 20.8.2010. Haettu 21.9.2012.

[59] Zakas, N. *Professional JavaScript for Web Developers*. John Wiley & Sons, 2011.

[60] Zalewski, M. Browser security handbook. *Google Code* (2010).